COMPILER ARCHITECTONICS

SCIENTIFIC PRINCIPLES AND PRACTICAL TECHNIQUES IN COMPILER DESIGN

by Fateha hossain Anushka

Compiler Architectonics

Scientific Principles & Practical Techniques in Compiler Design

By **Fateha Hossain Anushka**

Compiler Architectonics

Scientific Principles & Practical Techniques in Compiler Design

By **Fateha Hossain Anushka**

ID: 0242310005101844

Section: 64_C1

Self Publisher

This handbook, **"Compiler Architectonics : Scientific Principles & Practical Techniques in Compiler Design"**, is solely authored by **Fateha Hossain Anushka** and is intended strictly for academic and self-learning purposes. No part of this book may be copied, reproduced, distributed, or used in any form—digital or printed—for commercial, instructional, or any other unauthorized purposes without explicit written permission from the author.
This material is created as a personal reference guide for students studying compiler design. Any misuse, modification, or redistribution of this work is strictly prohibited.
For permission or inquiries, please contact:
anushka2305101844@diu.edu.bd

Copyright © 2025 Fateha Hossain Anushka

Preface

The field of Compiler Design stands as one of the most intricate and intellectually rewarding areas of Computer Science. A compiler is not just a program that translates high-level code into machine-understandable instructions; it is a remarkable orchestration of algorithms, data structures, and mathematical principles that together make modern computing possible. Yet for many students, the journey into compiler design often feels overwhelming due to the sheer depth and breadth of the subject.

This book, CompilerArchitectonics: Scientific Principles and Practical Techniques in Compiler Design, has been carefully crafted to bridge the gap between theory and practice, presenting the core concepts of compiler construction in a manner that is both deeply informative and approachable for students. My objective throughout this work has been to create a handbook that not only explains each phase of a compiler in detail but also illustrates these phases with clear examples, diagrams, and step-by-step C programming problems that solidify understanding.

The structure of this book reflects the natural flow of a compiler's operation, beginning with the fundamentals of lexical analysis and context-free grammars, progressing through finite automata, parsing techniques, syntax-directed translation, and intermediate code generation, and culminating in advanced topics such as control flow analysis and optimization techniques. Each chapter is designed to build upon the last, ensuring a smooth transition from the basics to more advanced compiler topics.

What makes this handbook unique is its emphasis on practical implementation alongside theory. For every major concept, I have included C programming exercises that replicate real compiler tasks — from validating identifiers and generating FIRST and FOLLOW sets, to constructing LR(0) parsing tables and producing optimized intermediate code. Every problem is provided with input and output examples, allowing readers to verify their understanding in practice.

This book was completed under the kind guidance and supervision of Mr. Mushfiqur Rahman Sir, whose encouragement, insightful feedback, and continuous support have been instrumental in shaping the quality and depth of this work. His mentorship has inspired me not only to learn the mechanics of compiler design but also to appreciate its elegance and significance in the field of

Computer Science.

It is my sincere hope that this handbook will serve as a dependable companion for students studying Compiler Design. Whether used for classroom learning, self-study, or reference during projects, the book is structured to be both comprehensive and accessible. By blending theoretical explanations with practical coding exercises, it strives to make the subject less daunting and more engaging, enabling learners to truly grasp the architecture of a compiler.

Finally, I extend my heartfelt gratitude to my peers, seniors, and the academic environment at Daffodil International University, which provided me the resources and motivation to accomplish this work.

I hope this book guides you with clarity and inspires you with curiosity as you embark on your own journey through the fascinating world of compiler construction.

— Fateha Hossain Anushka
Student of Department of CSE
ID:0242310005101844
Daffodil International University

Table of Contents

I Core Concepts of Compiler Design **1**

1 Introduction to Compiler Design **3**
1.1 What Is a Compiler? . . . 3
1.2 Characteristics of a Compiler . . . 4
1.3 Compiler vs Interpreter . . . 4
1.4 The Language Processing System . . . 4
1.4.1 Preprocessor . . . 5
1.4.2 Compiler . . . 5
1.4.3 Assembler . . . 6
1.4.4 Linker . . . 6
1.4.5 Loader . . . 7
1.5 Why Compiler Design Matters . . . 7
1.6 Role of Compiler in Execution . . . 7
1.7 Compiler Architecture Overview . . . 8
1.8 Summary . . . 8
1.9 C Programs . . . 8

2 Phases of a Compiler **13**
2.1 Introduction to Compiler Phases . . . 13
2.2 Six Major Phases of Compilation . . . 13
2.3 Phase 1: Lexical Analysis (Scanning) . . . 14
2.4 Phase 2: Syntax Analysis (Parsing) . . . 15
2.5 Phase 3: Semantic Analysis . . . 15
2.6 Phase 4: Intermediate Code Generation . . . 16
2.7 Phase 5: Code Optimization . . . 16
2.8 Phase 6: Code Generation . . . 17
2.9 Supporting Tools Inside the Compiler . . . 17
2.9.1 Symbol Table Manager . . . 17
2.9.2 Error Handler . . . 18
2.10 Lexical → Syntax → Semantic(Phase Chain in Action) . . . 18
Phase 1: Lexical Analysis (Tokenization) . . . 18
Phase 2: Syntax Analysis (Parsing) . . . 19
Phase 3: Semantic Analysis . . . 19

2.11 Intermediate Code → Optimization → Code Generation 20
Phase 4: Intermediate Code Generation 20
Phase 5: Code Optimization 21
Phase 6: Code Generation 21
2.12 Compiler Error Handling 22
2.12.1 Types of Errors in Compilation 22
2.12.2 Lexical Errors 22
2.12.3 Syntax Errors 22
2.12.4 Semantic Errors 22
2.13 Error Handling Strategies in Compilers 23
2.13.1 Error Handling Strategies Overview 23
2.13.2 Panic Mode Recovery 23
2.13.3 Phrase-Level Recovery 24
2.13.4 Error Productions 25
2.13.5 Global Correction 26
2.13.6 How to Recognize Which Strategy is Being Used 26
2.13.7 Best Practice Summary 27
2.14 Complete Example 27
2.15 Summary Table of Compiler Phases 29
2.16 C programs 29

3 Lexical Analysis & CFG **33**
3.1 Introduction to Lexical Analysis 33
3.2 Lexical Tokens, Patterns, Lexemes 33
3.3 Types of Tokens 34
3.4 Errors in Lexical Analysis 35
3.5 Introduction to CFG 35
3.5.1 Example CFG for Arithmetic Expressions 36
3.6 Parse Trees, LMD vs RMD 37
3.6.1 Examples 37
3.6.2 Practice Problems (With Hints) 41
3.7 Ambiguity in Grammar 41
3.7.1 Example of Ambiguous Grammar: 42
3.7.2 Example of Unambiguous Grammar 42
3.8 Examples & Practices 43
3.9 C Programming Problems 43

4 Regular Expressions **47**
4.1 Introduction to Regular Expressions 47
4.2 Basic Building Blocks of Regular Expressions 48
4.2.1 Basic Elements 48
4.2.2 Regular Expression Operators 48
4.3 Describing Tokens with REs 49
4.3.1 Practice – Classify These as Identifiers or Not 50
4.4 RE to Language 50

4.5 Language to RE . . . 52
4.6 Transition Diagrams . . . 53
4.6.1 Examples . . . 53
4.7 Limitations of Regular Expressions . . . 54
4.8 Practice Problems . . . 54
4.9 Summary . . . 55
4.10 C Programs Matching RE Patterns . . . 55

5 Finite Automata – NFA and DFA 61
5.1 Introduction to Finite Automata . . . 61
5.2 Types of Finite Automata . . . 61
5.3 Formal Definition of Finite Automaton . . . 62
5.4 DFA Concepts & Diagrams . . . 63
5.5 NFA Concepts and Diagrams . . . 64
5.6 Conversion from NFA to DFA . . . 64
Algorithm: Subset Construction . . . 65
5.7 DFA vs NFA: In-Depth Comparison . . . 66
5.8 Practice Problems . . . 67
5.9 Summary . . . 67
5.10 C programming Problems . . . 67

6 Left Recursion & Left Factoring 71
6.1 What Is Left Recursion? . . . 71
6.2 Eliminating Immediate Left Recursion . . . 72
6.3 Eliminating Indirect Left Recursion . . . 73
6.4 Introduction to Left Factoring . . . 74
6.5 Left Recursion vs Left Factoring . . . 75
6.6 Practice Problems . . . 75
6.7 Summary . . . 76
6.8 C programming Problems . . . 76

7 FIRST and FOLLOW Sets 79
7.1 Introduction: Why FIRST and FOLLOW Matter . . . 79
7.2 FIRST Set – Definition and Purpose . . . 79
7.3 FOLLOW Set - Definition and Purpose . . . 81
7.4 Combined Table for FIRST and FOLLOW . . . 83
7.5 How FIRST and FOLLOW Are Used in LL(1) Parsers . . . 83
7.6 Special Cases . . . 84
7.7 LL(1) Grammar Validity Test Using FIRST/FOLLOW . . . 84
7.8 Practice Exercises . . . 85
7.9 Summary Table . . . 86
7.10 C programming Problems . . . 86

8 LR(0) Parser and Canonical Table 89
8.1 Introduction to Bottom-Up Parsing . . . 89
8.2 What is an LR(0) Item? . . . 90
8.3 Augmented Grammar – Why and How . . . 90
8.4 Constructing LR(0) Items and States (With Examples) . . . 91
8.5 Parsing Table (ACTION, GOTO) . . . 92
8.6 String Parsing Example (Step-by-Step) . . . 93
8.7 Sketching LR(0) State Transition Diagram . . . 93
8.8 Conflict Detection in LR(0) . . . 94
8.9 LR(0) vs Other Parsers . . . 95
8.10 Practice Exercises . . . 95
8.11 Summary . . . 96
8.12 C Programming Problems . . . 96

II Intermediate Representation & Optimization 99

9 Intermediate Code Generation 101
9.1 Introduction to Intermediate Code Generation . . . 101
9.2 Desirable Properties of a Good Intermediate Representation . . . 102
9.3 Forms of Intermediate Representations . . . 102
9.4 Types of Three Address Code Instructions . . . 103
9.5 TAC Generation for Arithmetic Expressions . . . 104
9.6 Boolean Expressions and Conditional TAC . . . 104
9.7 Flow Graphs and Basic Blocks . . . 104
9.8 DAG Construction in Depth . . . 105
9.9 Large Worked-Out Example . . . 105
9.10 Practice Exercises . . . 105
9.11 Chapter Summary . . . 106
9.12 C Programming Problems . . . 106

10 Control Flow Analysis 109
10.1 Introduction . . . 109
10.2 Basic Blocks . . . 109
10.3 Leaders: The Entry Points . . . 111
10.4 Control Flow Graph (CFG) in Detail . . . 111
10.5 Properties of Control Flow Graph . . . 112
10.6 Loop Detection and Natural Loops . . . 112
10.7 Data Flow Analysis: Overview . . . 113
10.8 Dominators . . . 113
10.9 Post-Dominators . . . 114
10.10Control Dependence . . . 114
10.11Exception Handling in CFA . . . 114
10.12Summary and Exercises . . . 114
10.13C Programming Problems . . . 115

11 Code Optimization **117**
11.1 Introduction . . . 117
11.2 Objectives of Code Optimization . . . 118
11.3 Classification of Optimizations . . . 118
11.3.1 Machine-Independent Optimizations . . . 119
11.3.2 Machine-Dependent Optimizations . . . 119
11.3.3 Local vs. Global Optimizations . . . 119
11.4 Common Machine-Independent Optimizations . . . 119
11.4.1 Constant Folding . . . 119
11.4.2 Constant Propagation . . . 120
11.4.3 Dead Code Elimination . . . 120
11.4.4 Common Subexpression Elimination (CSE) . . . 120
11.4.5 Strength Reduction . . . 120
11.4.6 Loop Invariant Code Motion . . . 120
11.4.7 Loop Unrolling . . . 121
11.5 Peephole Optimization (In-Depth) . . . 121
11.6 Large Worked Example . . . 122
11.7 Exercises . . . 122
11.8 Summary . . . 123
11.9 C Programming Problems . . . 123

III Compiler Lab Project Toolkit **125**

12 Project: *Compiler Toolkit* **127**
12.1 Project Overview . . . 127
12.2 Motivation . . . 127
12.3 Features . . . 128
12.3.1 FIRST and FOLLOW Computation . . . 128
12.3.2 Left Recursion Detection & Elimination . . . 129
12.3.3 Left Factoring . . . 129
12.3.4 Regular Expression Matching . . . 129
12.3.5 Three-Address Code (TAC) Generation . . . 129
12.3.6 Custom Grammar & Input Handling . . . 130
12.4 Interface Snapshots . . . 130
12.5 Tools/Technologies Used . . . 132
12.6 GitHub Repository / Deployment Link . . . 133
12.7 Sample Use Cases . . . 134
12.8 Flowchart / Diagram of Process . . . 134

IV Lab Work Compilation **135**

13 Lab Sheet Solutions (With C Code) **137**
13.1 Lab Sheet 1: Program to Count the Length of a String . . . 137

13.2 Lab Sheet 2: Count the Number of White Spaces in a String . . 137
13.3 Lab Sheet 3: Remove White Spaces from a String 138
13.4 Lab Sheet 4: Multiple Line Input & Output 138
13.5 Lab Sheet 5: Extract Single Line Comments from String 139
13.6 Lab Sheet 6: Count Number of Lines 139
13.7 Lab Sheet 7: Tokenize Input String (Without strtok) 140
13.8 Lab Sheet 8: Identify and Count Articles 141
13.9 Lab Sheet 9: Identifier Validity and Keyword Check 142
13.10Lab Sheet 10: FIRST() of Two Productions (C++) 144
13.11Lab Sheet 11: Preposition Count 147
13.12Lab Sheet 12: Max Frequency Word 148
13.13Lab Sheet 13: RE Acceptance 149

A C Programming Essentials for Compiler Design 153
A.1 Strings, Arrays, Pointers Basics 153
A.2 Common Input Techniques 155
A.2.1 1. Character Input 155
A.2.2 2. String Input 156
A.2.3 3. Line Input 156
A.2.4 4. File Input 156
A.3 Manual vs Library Functions 156
A.3.1 Library Functions 157
A.3.2 Manual Implementation 157
A.3.3 In Compilers 157

B Reference Tables & Charts 159
B.1 ASCII Table 159
B.2 Regex Symbol Table 159
B.3 Grammar Notations 159
B.4 Token Classifications 159
B.5 Compiler Phases Recap 160

Glossary 161

Bibliography 163

Dedication

To My Mother,

For every silent prayer, Every warm meal on tired nights, For believing in me when I doubted myself, And for being my first and forever teacher— This book is a reflection of your strength, Bound with love, And dedicated to you, Who compiled my life with care.

Acknowledgements

I would like to express my sincere gratitude to **Mushfiqur Rahman** Sir, my Compiler Design Lab course instructor, for his invaluable guidance, patience, and insightful teaching throughout the semester. This handbook is a result of the foundation built in their lectures and lab sessions.

I am also thankful to my classmates and lab partners for their collaboration and encouragement during lab activities, brainstorming sessions, and problem-solving challenges.

Finally, I am grateful to my family and friends for their constant support, belief, and motivation in all phases of my academic journey.

Part I

Core Concepts of Compiler Design

Chapter 1

Introduction to Compiler Design

1.1 What Is a Compiler?

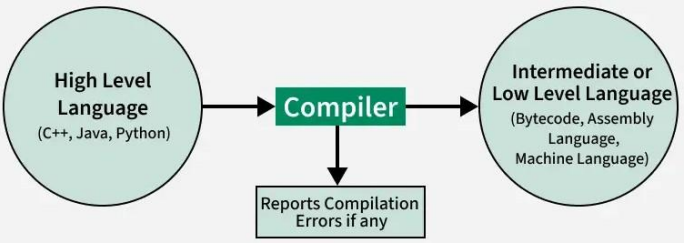
A compiler is a special program that translates high-level source code into machine-readable code so that the computer can understand and execute it. High-level programming languages like C, Java, and Python are human-friendly but not understandable by the CPU.

Why do we need a compiler?

- Computers only understand machine language — binary instructions like 10101010.
- Humans write programs in readable languages like:

```
int sum = 5 + 10;
```

- The compiler translates this human-readable code into a format that can be understood and executed by the CPU.



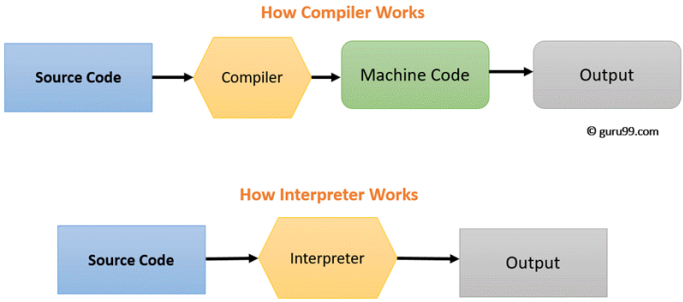
1.2 Characteristics of a Compiler

- **Complete translation:** Translates the whole program before execution.
- **Fast execution:** Since compiled code is pre-translated, it runs faster.
- **Optimization:** Reduces memory usage and speeds up code.
- **Error checking:** Identifies syntax and semantic errors during compilation.

1.3 Compiler vs Interpreter

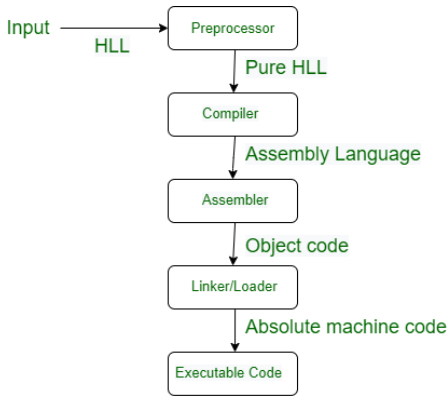
Feature	Compiler	Interpreter
Translation method	Entire program at once	Line-by-line
Speed of execution	Faster	Slower
Output	Object/machine code	No machine code
Error reporting	Shows all errors at once	Stops at first error
Language examples	C, C++, Java	Python, JavaScript, Ruby

Table 1.1: Comparison between Compiler and Interpreter



1.4 The Language Processing System

Before a high-level program becomes machine code, it goes through several stages:



1.4.1 Preprocessor

- Handles macro expansion, header file inclusion, and removal of comments before compilation starts.
- The output of the preprocessor is an expanded source file.

Example:

```
#include <stdio.h>
#define PI 3.1416
int main() {
    float area = PI * 5 * 5;
    return 0;
}
```

After pre-processing, the code becomes:

```
    int main() {
    float area = 3.1416 * 5 * 5;
    return 0;
}
```

1.4.2 Compiler

Purpose: The compiler takes the preprocessed source and performs:

- Lexical analysis (tokenization)
- Syntax analysis (parsing)
- Semantic analysis (meaning checks)
- Intermediate code generation (usually in the form of three-address code or abstract syntax trees)

Input: Preprocessed source file
Output: Assembly code file
Example:
Source:

```
int sum = 5 + 10;
```

Intermediate Code (Three-address style):

```
LDC R1, 5
LDC R2, 10
ADDF R3, R1, R2
STF sum, R3
```

This intermediate representation is what the compiler produces before actual machine code generation.

1.4.3 Assembler

The assembler translates the human-readable assembly code generated by the compiler into machine-level object code, which is in binary format.
Input: Assembly code file
Output: Object code file
Explanation:
Each line of the intermediate code is mapped to binary instructions that the CPU can execute. These binaries are stored in an object file.

1.4.4 Linker

The linker combines the object code with other necessary object files and libraries to produce a complete executable file.
Tasks performed by the linker:

- Resolves external references (e.g., function calls like printf)
- Links multiple object files
- Adds required library code

Input: One or more object files and libraries
Output: Executable file
Example:
If your program uses sqrt(), the linker connects your object file with the math library containing sqrt() implementation.

1.4.5 Loader

The loader is part of the operating system. It loads the executable into main memory, assigns addresses, and prepares it for execution.

Input: Executable file
Output: Running process in memory
Tasks performed by the loader:

- Allocates memory space
- Sets up the stack, heap, and code segment
- Loads libraries and runtime dependencies
- Passes control to the program's starting point (main() in C)

1.5 Why Compiler Design Matters

- Helps you understand how your code is executed
- Useful in building your own programming languages
- Essential in creating development tools (IDEs, linters, debuggers)
- Critical in system security (code analysis)
- Fundamental for optimizing software performance

1.6 Role of Compiler in Execution

Suppose you write:

```
int result = 3 * 4 + 2;
```

Step-by-step actions:
Tokenization: int, result, =, 3, *, 4, +, 2, ;
Syntax Tree:

```
    +
   / \
  *   2
 / \
3   4
```

Intermediate Code:

```
LDC R1, 3
LDC R2, 4
MULF R3, R1, R2
LDC R4, 2
ADDF R5, R3, R4
STF result, R5
```

1.7 Compiler Architecture Overview

The compiler is divided into two main parts:

Front-End

- Lexical Analysis (scans and creates tokens)
- Syntax Analysis (parses using grammar)
- Semantic Analysis (type checks)
- Intermediate Code Generator

Back-End

- Code Optimizer
- Target Code Generator
- Code Emission

1.8 Summary

- A compiler is a multi-phase translator that converts high-level code to machine code.
- The language processing system includes preprocessing, compiling, assembling, linking, and loading.
- Intermediate code generation helps in optimization and portability.
- Understanding compilation deepens your grasp of how programs execute and how languages work internally.

1.9 C Programs

Problem 1: Identify whether a string is a valid identifier

Problem Statement: Write a C program that checks if the given input string is a valid identifier according to simple rules:

- Must start with a letter or underscore
- Followed by letters, digits, or underscores
- No spaces or special symbols

C Code

```
#include <stdio.h>

int main() {
    char str[100];
    int i = 0, flag = 1;
    printf("Enter a string: ");
    scanf("%s", str);

    // First character check
    if (!((str[0] >= 'A' && str[0] <= 'Z') ||
          (str[0] >= 'a' && str[0] <= 'z') ||
           str[0] == '_')) {
        flag = 0;
    }

    // Remaining characters check
    while (str[i] != '\0') {
        if (!((str[i] >= 'A' && str[i] <= 'Z') ||
              (str[i] >= 'a' && str[i] <= 'z') ||
              (str[i] >= '0' && str[i] <= '9') ||
               str[i] == '_')) {
            flag = 0;
            break;
        }
        i++;
    }

    if (flag)
        printf("Valid Identifier\n");
    else
        printf("Invalid Identifier\n");

    return 0;
}
```

Input & Output

```
Input:  abc123
Output: Valid Identifier
```

```
Input:  1abc
Output: Invalid Identifier
```

Problem 2: Count Number of Keywords in a Program

Problem Statement: Write a C program to count how many keywords are present in a given C code snippet.

C Code

```
#include <stdio.h>
#include <string.h>

int main() {
    char keywords[10][10] = {"int","float","if","else","while","for","return",
    "char","void","main"};
    char word[50];
    int count = 0, i, j, found;
    printf("Enter words (end with 0):\n");

    while (1) {
        scanf("%s", word);
        if (strcmp(word,"0")==0) break;

        found = 0;
        for (i=0; i<10; i++) {
            if (strcmp(word,keywords[i])==0) {
                count++;
                found = 1;
                break;
            }
        }
    }
    printf("Number of Keywords: %d\n", count);
    return 0;
}
```

Input & Output

```
Input:
int a b if else 0
Output:
Number of Keywords: 3
```

Problem 3: Simple Lexical Token Separator

Problem Statement: Write a C program to separate identifiers, numbers, and operators from an expression.

C Code

```
#include <stdio.h>
#include <ctype.h>

int main() {
    char expr[100];
    int i=0;
    printf("Enter an expression: ");
    scanf("%s", expr);

    while (expr[i] != '\0') {
        if (isalpha(expr[i])) {
            printf("Identifier: %c\n", expr[i]);
        }
        else if (isdigit(expr[i])) {
            printf("Number: %c\n", expr[i]);
        }
        else {
            printf("Operator: %c\n", expr[i]);
        }
        i++;
    }
    return 0;
}
```

Input & Output

```
Input: a+3*b
Output:
Identifier: a
Operator: +
Number: 3
Operator: *
Identifier: b
```

Chapter 2

Phases of a Compiler

2.1 Introduction to Compiler Phases

When we write a program, we think in terms of logic, variables, loops, and conditions. But a compiler doesn't "understand" our code directly. It processes the code through several well-defined stages, each performing a unique task to transform our high-level source code into low-level machine code.

These stages are called the phases of a compiler, and each phase takes input from the previous phase and prepares output for the next.

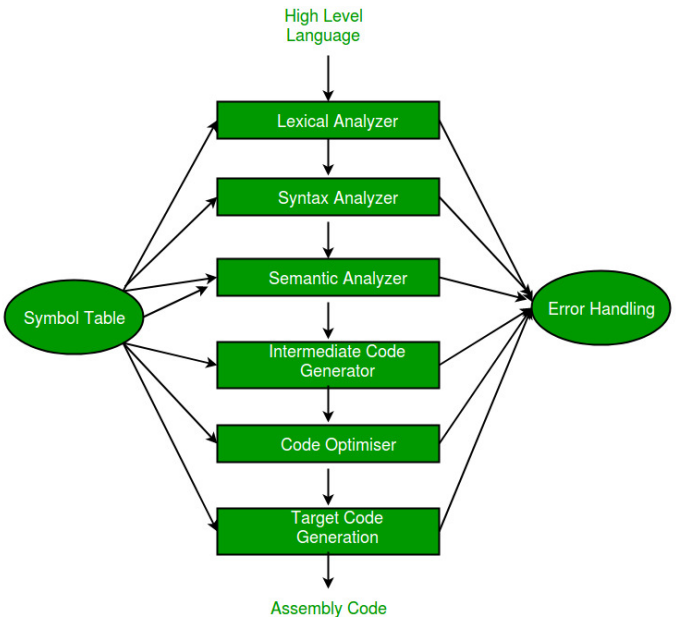
2.2 Six Major Phases of Compilation

The six essential phases are:

1. Lexical Analysis
2. Syntax Analysis
3. Semantic Analysis
4. Intermediate Code Generation
5. Code Optimization
6. Code Generation

There are also **two important helper** components:

1. Symbol Table Manager
2. Error Handler



Let's explore each phase deeply with examples.

2.3 Phase 1: Lexical Analysis (Scanning)

What it does: Breaks the source code into tokens—the smallest meaningful units like keywords, identifiers, numbers, symbols, and operators.

Input: Raw source code (e.g., int result = a + b;)

Output: Stream of tokens (int, result, =, a, +, b, ;)

Example 1: Input string:

```
float total = price + tax;
```

Lexical Analyzer (also called scanner) breaks it into:

```
[float] [total] [=] [price] [+] [tax] [;]
```

Each word is categorized as:

- float → keyword
- total, price, tax → identifiers
- = and + → operators
- ; → delimiter

If there's a typo like @price, the lexical analyzer reports a lexical error.

2.4 Phase 2: Syntax Analysis (Parsing)

What it does:

Checks the structure of the token stream using rules of grammar, and builds a parse tree or syntax tree.

Input: Stream of tokens
Output: Parse Tree or Abstract Syntax Tree (AST) **Example 2:**
Input tokens:

```
[total] [=] [price] [+] [tax]
```

The parser checks:

- Is this a valid assignment expression?
- Are the operands and operators in the correct order?

Tree Representation:

```
      =
     / \
total   +
       / \
   price  tax
```

If a rule is violated (e.g., = price + tax total), it raises a syntax error.

2.5 Phase 3: Semantic Analysis

What it does:

Verifies that meaningful associations are correct. It checks:

- Are variables declared?
- Are data types compatible?
- Are operations legal?

Input: Parse tree or AST
Output: Annotated tree or semantic error
Example 3:
Input:

```
int a = "apple";
```

Syntactically, this is correct. But semantically, it's wrong because you cannot assign a string to an integer.

Another example:

```
sum = rate + quantity;
```

If sum and rate are declared but quantity is not, a semantic error is thrown: "Undefined variable: quantity".

2.6 Phase 4: Intermediate Code Generation

What it does:

Generates intermediate code (IC) that sits between high-level and machine code. This is portable and easier to optimize.

Input: Annotated syntax tree

Output: Intermediate representation (IR)

Example 4:

Input:

```
amount = price + tax
```

Intermediate code (three-address code style):

```
t1 = price + tax
amount = t1
```

Another format using compiler-like notation:

```
LDF R1, price
LDF R2, tax
ADDF R3, R1, R2
STF amount, R3
```

This stage breaks complex expressions into simple, sequential operations.

2.7 Phase 5: Code Optimization

What it does: Improves the intermediate code by removing redundancy, minimizing instructions, and enhancing speed and efficiency—without changing output.

Input: Intermediate code

Output: Optimized intermediate code

Example 5: Unoptimized:

```
t1 = a + b
t2 = a + b
t3 = t1 * t2
```

Optimized:

```
t1 = a + b
t3 = t1 * t1
```

Here, we avoided repeating the same addition. **Another example:**

```
t1 = x * 1    →   t1 = x   (Multiplication by 1 removed)
t2 = y + 0    →   t2 = y   (Addition by 0 removed)
```

This phase is critical in real-world compilers for making programs fast and compact.

2.8 Phase 6: Code Generation

What it does:

Translates the optimized intermediate code into machine-level code or assembly code that the processor can execute.

Input: Optimized IR

Output: Final assembly or object code

Example 6: IR:

```
t1 = price + tax
total = t1
```

Final code:

```
LDF R1, price
LDF R2, tax
ADDF R3, R1, R2
STF total, R3
```

Registers are used to hold temporary results and the final result is stored in memory.

2.9 Supporting Tools Inside the Compiler

2.9.1 Symbol Table Manager

Keeps track of:

- Variable names
- Data types
- Scope (global or local)
- Memory location

Each phase accesses this table to validate variable use.

2.9.2 Error Handler

Handles:

- Lexical errors (e.g., invalid character)
- Syntax errors (e.g., missing ;)
- Semantic errors (e.g., type mismatch)
- Runtime errors (e.g., divide by zero)

All errors are reported with line numbers and messages.

2.10 Lexical → Syntax → Semantic(Phase Chain in Action)

Let's walk through every phase using a real example and show what each phase does with diagrams and logic.

Input Code:

```
int result = a + b * 2;
```

Phase 1: Lexical Analysis (Tokenization)

Goal: Convert source code into a sequence of tokens.

Token Stream:

```
[int] [result] [=] [a] [+] [b] [*] [2] [;]
```

Token Categories:

- int → Keyword
- result, a, b → Identifiers
- = → Assignment Operator
- +, * → Arithmetic Operators
- 2 → Numeric Constant
- ; → Semicolon (Delimiter)

If a token like @b was found, it would be a lexical error: invalid character.

Phase 2: Syntax Analysis (Parsing)

Goal: Use grammar rules to form a syntax tree (parse tree) and check for correct structure.

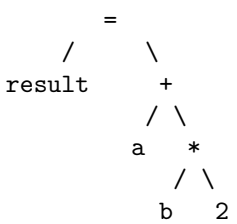
Sample Grammar Rules:

```
S → Type ID = E ;
E → E + T | T
T → T * F | F
F → ID | NUM
```

Step-by-Step Derivation (Leftmost):

```
S
→ int ID = E ;
→ int result = E ;
→ int result = E + T ;
→ int result = T + T ;
→ int result = F + T ;
→ int result = a + T ;
→ int result = a + T * F ;
→ int result = a + b * F ;
→ int result = a + b * 2 ;
```

Parse Tree:



If the input was a = + b, the parser would throw a syntax error: operator without left operand.

Phase 3: Semantic Analysis

Goal: Check for meaning correctness:

- Are variables declared?
- Are data types compatible?
- Are values initialized?

Checks for Our Example:

- Is a declared?
- Is b declared?
- Is result declared and of type int?
- Are all expressions type-consistent?

Semantic Errors:

```
int name = "John";      // type mismatch (string to int)
int z = x + y;          // if x or y undeclared → undefined variable error
```

Semantic errors are only detectable after the syntax is correct.

2.11 Intermediate Code → Optimization → Code Generation

Phase 4: Intermediate Code Generation

Goal: Create low-level abstract operations using registers and temporary variables.

Input:

```
result = a + b * 2
```

Intermediate Code (Three Address Code):

```
t1 = b * 2
t2 = a + t1
result = t2
```

In compiler-like operations:

```
LDF R1, b
LDC R2, 2
MULF R3, R1, R2
LDF R4, a
ADDF R5, R4, R3
STF result, R5
```

Phase 5: Code Optimization

Goal: Remove redundancy or simplify expressions.

Before Optimization:

```
t1 = b * 2
t2 = a + t1
result = t2
```

Optimization Opportunities:

- If b = 10, then b * 2 = 20 → Constant Folding
- If t1 was already computed elsewhere → Eliminate Recalculation

After Optimization:

```
t2 = a + (b * 2)
result = t2
```

Phase 6: Code Generation

Final machine-like code generation:

```
LDF R1, b
LDC R2, 2
MULF R3, R1, R2
LDF R4, a
ADDF R5, R4, R3
STF result, R5
```

This is now ready for the assembler to convert to object code.

2.12 Compiler Error Handling

2.12.1 Types of Errors in Compilation

Error Type	Detected By	Example
Lexical Error	Lexical Analyzer	Invalid characters (e.g., ‘@x‘)
Syntax Error	Syntax Analyzer	Missing operator (e.g., ‘a + ;‘)
Semantic Error	Semantic Analyzer	Undeclared variable (e.g., ‘x + y‘, where ‘y‘ is unknown)
Logical Error	Not by Compiler	Wrong output due to faulty logic
Runtime Error	OS/Runtime System	Division by zero, memory access errors

Table 2.1: Types of Errors in Compilation

2.12.2 Lexical Errors

Detected during tokenization.
Examples:

```
Invalid characters ($name)
Unfinished string literal ("hello)
Misplaced numbers (1var)
```

2.12.3 Syntax Errors

Detected during parsing.
Examples:

- Missing semicolon
- Unmatched parentheses
- Wrong token sequence (= + a)

2.12.4 Semantic Errors

Detected during meaning check.
Examples:

- Type mismatch (int x = ”hello”;)

- Undeclared identifiers
- Function used with wrong arguments

2.13 Error Handling Strategies in Compilers

A good compiler doesn't stop on the first error—it tries to recover and continue processing the rest of the code. This way, the programmer gets to know all the mistakes in one go, not just the first one.

To do this, compilers use error handling strategies, which are special techniques to detect, report, and recover from errors gracefully.

2.13.1 Error Handling Strategies Overview

Strategy	**Idea**	**Example Situation**
Panic Mode Recovery	Skip tokens until a known good point is found	Skip till ';' after syntax error
Phrase-Level Recovery	Make small local corrections to continue parsing	Insert missing token like ';' or ')'
Error Productions	Add rules in grammar to handle common errors	Special 'error' token in grammar
Global Correction	Suggest minimal changes to make the input correct	IDEs suggesting auto-correction

Table 2.2: Error Recovery Strategies in Parsing

2.13.2 Panic Mode Recovery

Definition:

The compiler discards input symbols until it finds a synchronization point, like ;, , or —a point where parsing can safely resume.

How It Works:

- Error is found
- Skip tokens until a known delimiter (sync token)
- Resume parsing from next valid construct

Example:

```
int x = 5
int y = 10;
```

Missing ; after x = 5 will cause panic mode to skip tokens until int is seen, assuming it's a new declaration.

Safe but may skip too much.

What Compiler Does:

- Detects missing ;
- Skips till int
- Resumes from int y = 10;

When to Use:

- When code is messy
- To prevent infinite error chains

Advantages:

- Simple and fast
- Doesn't get stuck

Disadvantages:

- May skip too much
- Can miss minor fixes

2.13.3 Phrase-Level Recovery

Definition: The compiler locally fixes the error by inserting, deleting, or replacing a few symbols.

How It Works:

- Detects an unexpected token
- Tries to fix using grammar knowledge
- Resumes parsing

Example:

```
int x = ;       // invalid assignment
```

Compiler Might:

- Insert 0 as a default value:

```
int x = 0;
```

When to Use:

- Minor syntax mistakes
- Safe for compiler-generated fixes

Advantages:

- Keeps parsing accurate
- Minimal disruption

Disadvantages:

- May change original logic
- More complex to implement

2.13.4 Error Productions

Definition: The grammar itself includes rules for common mistakes so the parser can recognize and report them.

How It Works: You add grammar like:

```
stmt → error ';'
```

If something goes wrong before a semicolon, it matches this rule and reports a statement error. **Example:**

```
int x = @5;     // invalid token
```

Compiler matches stmt → error ; and says: "Invalid statement near @5"

When to Use:

- Known predictable mistakes
- Student-friendly error messages

Advantages:

- Controlled and customized
- Clear error reports

Disadvantages:

- Requires manual error rule design
- Cannot catch all unknown errors

2.13.5 Global Correction

Definition: The compiler tries to find the minimum number of edits (insertions, deletions, substitutions) needed to make the code correct.

How It Works:

- Uses dynamic programming or AI
- Predicts best possible correction
- Suggests or applies it

Example:

```
int x ( 5 = ;   // many mistakes
```

Compiler might suggest:

```
int x = 5;
```

When to Use:

- Advanced compilers
- IDEs (e.g., Eclipse, IntelliJ)

Advantages:

- Smart and user-friendly
- Fixes deeply broken code

Disadvantages:

- Not used in real-time compilation
- Very costly (time and memory)

2.13.6 How to Recognize Which Strategy is Being Used

Scenario	**Likely Strategy Used**
Compiler skips multiple tokens after an error	Panic Mode Recovery
Compiler inserts or replaces a token	Phrase-Level Recovery
Parser recognizes ‘error‘ in grammar	Error Productions
IDE auto-corrects broken code	Global Correction

Table 2.3: Error Recovery Strategy by Scenario

2.13.7 Best Practice Summary

Strategy	Use When	Result
Panic Mode	Serious structure issues	Quick recovery, skips code
Phrase-Level	Simple, local mistakes	Accurate fix, keeps flow
Error Productions	Known frequent mistakes	Friendly error messages
Global Correction	IDE or AI-based tools	Full suggestions, high cost

Table 2.4: Comparison of Error Recovery Strategies

2.14 Complete Example

Let's walk through the full compilation of this input: **Example 1:a = b + c * d**

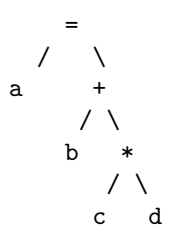
```
a = b + c * d
```

1. **Lexical Analysis** Input string: a = b + c * d
   Tokens identified:

Lexeme	Token
a	id (identifier)
=	assignment operator
b	id
+	plus operator
c	id
*	multiplication operator
d	id

Table 2.5: Lexemes and Corresponding Tokens

2. Syntax Analysis (Parsing)
Parse tree for a = b + c * d:



3. Semantic Analysis Check:

- Variables a, b, c, d are declared
- Type compatibility: assume all are float
- Valid assignment to a

No semantic error
Symbol Table:

Name	Type
a	float
b	float
c	float
d	float

Table 2.6: Variable Names and Their Types

4. Intermediate Code Generation (3AC)

```
t1 = MULF c, d      ; t1 = c * d
t2 = ADDF b, t1     ; t2 = b + t1
STOF t2, a          ; a = t2
```

5.Code Optimization Apply strength reduction, eliminate common subexpressions, etc.

- Already minimal
- No redundant expressions

Optimized 3AC remains:

```
t1 = MULF c, d
t2 = ADDF b, t1
STOF t2, a
```

6. Code Generation (Target Code)

```
LDF R1, c
LDF R2, d
MULF R3, R1, R2
LDF R4, b
ADDF R5, R4, R3
STOF a, R5
```

2.15 Summary Table of Compiler Phases

Phase	Description	Output
Lexical Analysis	Breaks source into tokens	Token stream
Syntax Analysis	Checks grammar structure	Parse tree / AST
Semantic Analysis	Validates meaning and types	Annotated tree / error
Intermediate Code	Creates low-level version of logic	Intermediate code (IC)
Code Optimization	Improves IC for speed and space	Optimized intermediate code
Code Generation	Converts to machine code	Assembly / object code

Table 2.7: Phases of Compilation with Descriptions and Outputs

2.16 C programs

Problem 1: Simulate Lexical → Syntax → Semantic

Statement: Program that simulates detection of identifiers, operators, and builds a simple parse-like output.

C Code

```
#include <stdio.h>
#include <ctype.h>
int main() {
    char expr[100]; int i=0;
    printf("Enter simple expression: ");
    scanf("%s", expr);
    printf("Lexical Tokens:\n");
    while (expr[i]!='\0') {
        if (isalpha(expr[i])) printf("ID(%c)\n", expr[i]);
        else if (isdigit(expr[i])) printf("NUM(%c)\n", expr[i]);
        else printf("OP(%c)\n", expr[i]);
        i++;
    }
    printf("Syntax: Expression Detected\n");
    printf("Semantic: Expression Type Checked\n");
    return 0;
}
```

Input/Output

```
Input: a+5
Output:
Lexical Tokens:
ID(a)
OP(+)
NUM(5)
Syntax: Expression Detected
Semantic: Expression Type Checked
```

Problem 2: Build Symbol Table

C Code

```
#include <stdio.h>
#include <string.h>
struct symbol { char name[20]; char type[10]; };
int main() {
    struct symbol table[10];
    int n,i;
    printf("Enter number of identifiers: ");
    scanf("%d", &n);
    for(i=0;i<n;i++) {
        printf("Enter name and type: ");
        scanf("%s %s", table[i].name, table[i].type);
    }
    printf("Symbol Table:\n");
    for(i=0;i<n;i++) printf("%s\t%s\n", table[i].name, table[i].type);
    return 0;
}
```

Input/Output

```
Input:
2
x int
y float
Output:
Symbol Table:
x int
y float
```

Problem 3: Error Type Recognition

C Code

```
#include <stdio.h>
#include <string.h>
int main() {
    char input[100];
    printf("Enter a line: ");
    scanf("%s", input);
    if (strchr(input,'$')) printf("Lexical Error: Invalid char\n");
    else if (strcmp(input,"if")==0) printf("Syntax Error:

    Incomplete statement\n");
    else printf("No Error Detected\n");
    return 0;
}
```

Input/Output

```
Input: ab$
Output: Lexical Error: Invalid char
```

Chapter 3

Lexical Analysis & CFG

3.1 Introduction to Lexical Analysis

Lexical analysis is the first stage of the compiler. Its job is to scan the source program and break it into a sequence of tokens (the smallest units of meaning, like keywords, identifiers, operators, and constants).

The Lexical Analyzer (scanner):

- Reads characters from the source file.
- Groups them into lexemes (valid sequences of characters).
- Produces tokens representing those lexemes.
- Removes unnecessary characters like whitespaces and comments.

Why Lexical Analysis?

- The compiler cannot process a raw source file directly because it is just a sequence of characters.
- Lexical analysis groups characters into meaningful tokens so that the syntax analyzer can build grammar structures.

3.2 Lexical Tokens, Patterns, Lexemes

Token: A classification label (e.g., keyword, identifier, constant).

Lexeme: The actual sequence of characters that match a token pattern.

Pattern: The rule that describes the set of possible lexemes.

Examples of Tokens and Lexemes

Example 1:

```
int sum = 100;
```

- int → keyword token
- sum → identifier token
- 100 → constant token
- = → operator token
- ; → delimiter token

Example 2:

```
float area = width * height;
```

- Tokens: float, area, =, width, *, height, ";"
- Lexemes: "float", "area", "=", "width", "*", "height", ";"

Example 3:

```
if (a > b) c = a;
```

- Tokens: if (keyword), ((delimiter), a (identifier), ¿ (operator), b (identifier),) (delimiter), c (identifier), = (operator), a (identifier), ; (delimiter)

Practice Task:
Break this line into tokens:

```
for (i = 0; i < 10; i++) sum = sum + i;
```

Tokens: for, (, i, =, 0, ;, i, ¡, 10, ;, i, ++,), sum, =, sum, +, i, ;

3.3 Types of Tokens

- **Keywords:** Predefined words like int, float, if, while.
- **Identifiers:** User-defined names like x, total, sum1.
- **Literals/Constants:** Numeric or string values like 100, "hello".
- **Operators:** Symbols like +, -, *, /, ==.
- **Separators:** ;, ,, (,), , .

Examples of Tokenization
Example 1:

```
int marks = 90;
→ [int] [marks] [=] [90] [;]
```

Example 2:

```
while (x <= 10) x = x + 1;
→ [while] [(] [x] [<=] [10] [)] [x] [=] [x] [+] [1] [;]
```

Example 3:

```
printf("Hello");
→ [printf] [(] ["Hello"] [)] [;]
```

Practice Task:
Break this line into tokens:

```
total = (price + tax) * discount;
```

3.4 Errors in Lexical Analysis

Lexical errors occur during tokenization when the source contains invalid characters or malformed tokens.

Examples of Lexical Errors:

```
1.  int $x = 5; → $ ->is not allowed in identifiers.
2.  char name = 'AB; → Unclosed character literal.
3.  3var = 2; → Identifiers cannot start with a number.
```

How to Handle Lexical Errors?

- Skipping invalid characters.
- Replacing invalid symbols with valid ones (if predictable).
- Using panic mode to skip to the next delimiter.

3.5 Introduction to CFG

A grammar defines the rules of a language, describing which strings are valid.

A Context-Free Grammar (CFG) is defined by:
G = (V, T, P, S)

- V: Variables (non-terminals)
- T: Terminals (tokens like +, id)
- P: Production rules
- S: Start symbol

3.5.1 Example CFG for Arithmetic Expressions

Grammar:

```
E → E + T | T
T → T * F | F
F → (E) | id
```

Meaning:

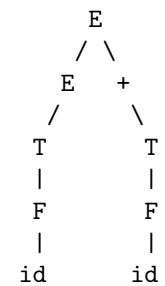
- E represents an expression.
- An expression is either E + T or just T.
- A term (T) is either T * F or just F.
- A factor (F) is either an identifier (id) or a parenthesized expression (E).

Example 1: Parsing id + id

Derivation (Leftmost):

```
  E
→ E + T
→ T + T
→ F + T
→ id + T
→ id + F
→ id + id
```

Parse Tree:



Example 2: Parsing id + id * id

Derivation:

```
    E
→ E + T
→ T + T
→ F + T
```

```
→ id + T
→ id + T * F
→ id + F * F
→ id + id * id
```

Example 3: Parentheses Parsing (id + id) * id
Steps:

```
  E
→ T
→ T * F
→ F * F
→ (E) * F
→ (E + T) * F
→ (T + T) * F
→ (F + T) * F
→ (id + T) * F
→ (id + F) * F
→ (id + id) * id
```

3.6 Parse Trees, LMD vs RMD

A **Parse Tree** is a tree structure that represents how a string is derived from a grammar.

- Leftmost vs Rightmost Derivation:
- Leftmost: Always expand the leftmost non-terminal first.
- Rightmost: Always expand the rightmost non-terminal first.

Both lead to the same final string, but the steps differ.
Why Are Derivations Important?
They help:

- Visualize how the grammar generates a string.
- Construct parse trees.
- Understand ambiguity in grammars

3.6.1 Examples

Grammar We'll Use For All Examples:

```
E → E + T | T
T → T * F | F
F → ( E ) | id
```

This grammar supports:

- id, id + id, id * id, (id + id) * id, etc.
- Operator precedence: * +
- Left-associativity of + and *

Example 1: Derive id + id

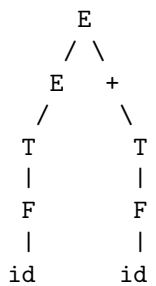
Leftmost Derivation (LMD):

```
    E
→ E + T        (by E → E + T)
→ T + T        (E → T)
→ F + T        (T → F)
→ id + T       (F → id)
→ id + F       (T → F)
→ id + id      (F → id)
```

Rightmost Derivation (RMD):

```
    E
→ E + T        (by E → E + T)
→ E + F        (T → F)
→ E + id       (F → id)
→ T + id       (E → T)
→ F + id       (T → F)
→ id + id      (F → id)
```

Parse Tree:



Example 2: Derive id + id * id

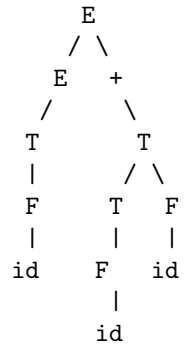
Leftmost Derivation (LMD):

```
    E
→ E + T
→ T + T
→ F + T
→ id + T
→ id + T * F
→ id + F * F
→ id + id * id
```

Rightmost Derivation (RMD):

```
    E
→ E + T
→ E + T * F
→ E + F * F
→ E + id * F
→ E + id * id
→ T + id * id
→ F + id * id
→ id + id * id
```

Parse Tree:



Note: The tree groups id * id first, meaning * has higher precedence than +.

Example 3: Derive (id + id) * id

Leftmost Derivation (LMD):

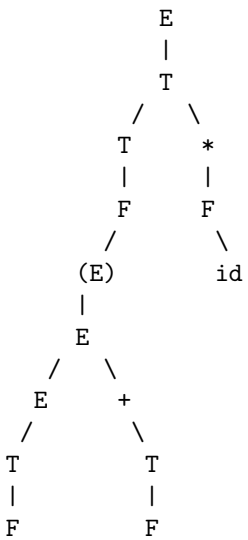
```
E
→ T
```

```
→ T * F
→ F * F
→ ( E ) * F
→ ( E + T ) * F
→ ( T + T ) * F
→ ( F + T ) * F
→ ( id + T ) * F
→ ( id + F ) * F
→ ( id + id ) * F
→ ( id + id ) * id
```

Rightmost Derivation (RMD):

```
E
→ T
→ T * F
→ F * F
→ ( E ) * F
→ ( E ) * id
→ ( E + T ) * id
→ ( T + T ) * id
→ ( F + T ) * id
→ ( id + T ) * id
→ ( id + F ) * id
→ ( id + id ) * id
```

Parse Tree:



```
 |          |
id         id
```

This tree clearly shows the expression inside parentheses is grouped first.

3.6.2 Practice Problems (With Hints)

- Given the grammar above, derive id * id + id using LMD and RMD.
- Hint: Think operator precedence.
- Build the parse tree for (id * id) + id
- Hint: The * operation is grouped first, then +.
- Find LMD for: (id + id) * (id + id)
- Is the grammar below ambiguous?

```
E → E + E | E * E | id
```

  Derive id + id * id and draw two parse trees to prove ambiguity.

3.7 Ambiguity in Grammar

What is Ambiguity?

A grammar is ambiguous if a single string can be generated by the grammar in more than one way—specifically:

- It has more than one leftmost derivation
- It has more than one rightmost derivation
- It has more than one parse tree

Ambiguity leads to confusion during parsing, because the compiler cannot determine the correct structure of a sentence.

Why Ambiguity is a Problem

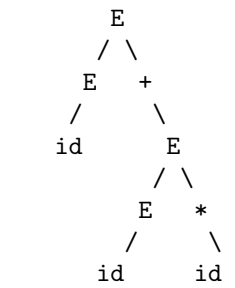
Ambiguity must be avoided in compiler design because:

- It causes conflicting interpretations
- Makes syntax analysis unreliable
- Affects code generation and optimization

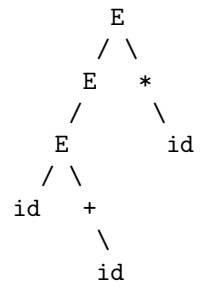
3.7.1 Example of Ambiguous Grammar:

```
E → E + E | E * E | id
```

String: **id + id * id**
Parse Tree 1: id + (id * id)



Parse Tree 2: (id + id) * id

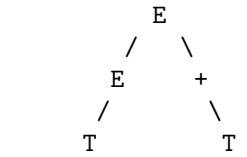


Two different parse trees for the same string, So **Grammar is ambiguous**.

3.7.2 Example of Unambiguous Grammar

```
E → E + T | T
T → T * F | F
F → (E) | id
```

Derive id + id * id: Only one parse tree will be generated (groups id * id first):



```
 |       / \
 F      T   F
 |      |   |
id     id  id
```

One unique tree , So **Grammar is unambiguous.**

3.8 Examples & Practices

Exercise 1: CFG for Balanced Parentheses Task:
Write a CFG that generates only balanced parentheses, such as:

- ””
- ()
- (())
- ()()
- (()())

Hint: Use recursion with empty string.
These exercises helps strengthen understanding of Context-Free Grammars (CFGs), derivations, parse trees, ambiguity detection, and grammar construction. **Exercise 2: Derivation Tree for Specific String** Given grammar:

```
S → aSb | ab
```

Task:
Draw the leftmost derivation, rightmost derivation, and the parse tree for the string:

```
aaabbb
```

Question:
Is the grammar ambiguous?

3.9 C Programming Problems

Problem 1: Count Tokens in a CFG Production

C Code

```
#include <stdio.h>
#include <string.h>
int main() {
    char prod[100];
    int i,count=0;
    printf("Enter CFG Production: ");
    scanf("%[^\n]", prod);
    for(i=0;prod[i]!='\0';i++)
        if(prod[i]==' '||prod[i]=='|'||prod[i]=='-') count++;
    printf("Number of Tokens: %d\n", count+1);
    return 0;
}
```

Input/Output

```
Input: S -> aA | b
Output: Number of Tokens: 5
```

Problem 2: Parse Tree Printer (Linear)

C Code

```
#include <stdio.h>
int main() {
    printf("Grammar: S->aSb|c\n");
    printf("String: acb\n");
    printf("Parse Tree:\n");
    printf("     S\n");
    printf("   / | \\\n");
    printf("  a  S  b\n");
    printf("     |\n");
    printf("     c\n");
    return 0;
}
```

Output

```
Grammar: S->aSb|c
String: acb
Parse Tree:
     S
   / | \
  a  S  b
     |
     c
```

Problem 3: Leftmost and Rightmost Derivation

C Code

```
#include <stdio.h>
int main() {
    printf("Grammar: S -> aSb | c\n");
    printf("String: acb\n");
    printf("Leftmost Derivation:\n");
    printf("S => aSb => acb\n");
    printf("Rightmost Derivation:\n");
    printf("S => aSb => acb\n");
    return 0;
}
```

Output

```
Grammar: S -> aSb | c
String: acb
Leftmost Derivation:
S => aSb => acb
Rightmost Derivation:
S => aSb => acb
```

Chapter 4

Regular Expressions

4.1 Introduction to Regular Expressions

In compiler design, regular expressions (REs) are used to describe the structure of tokens in a programming language. Tokens include keywords, identifiers, constants, operators, and punctuations. The lexical analyzer relies on REs to identify and extract these tokens from the source code.

A **regular expression** is a mathematical notation for defining regular languages, which are the simplest class of languages recognized by finite automata.

Regular expressions are tightly coupled with lexical analysis, the first phase of the compiler. Each token type in a programming language is described using an RE, which is later translated into finite automata for recognition.

Applications in Compiler Design

- Token classification (keywords, identifiers, numbers, etc.)
- Input string scanning and segmentation
- Foundation for building NFA/DFA
- Used in tools like Lex for lexical analyzer generation

Characteristics of Regular Languages

- Can be defined by REs
- Can be recognized by Finite Automata (FA)
- Cannot count or match nested patterns (like balanced parentheses) — those require context-free grammars

4.2 Basic Building Blocks of Regular Expressions

4.2.1 Basic Elements

Let's build an RE from the ground up using the alphabet = a, b.

Expression	Meaning
`a`	A string containing the symbol `a`
`b`	A string containing the symbol `b`
ε	The empty string (length zero)
	Empty language (matches nothing)

Table 4.1: Basic Regular Expressions and Their Meanings

4.2.2 Regular Expression Operators

Operator	Name	Meaning
`|`	Union / OR	`a|b` matches `a` or `b`
`.`	Concatenation	`ab` matches the string "ab"
`*`	Kleene Star	`a*` matches ε, `a`, `aa`, `aaa`, ... (zero or more repetitions)
`+`	One or More	`a+` matches `a`, `aa`, `aaa`, ... (at least once)
`?`	Optional	`a?` matches `a` or ε
`()`	Grouping	Used to group parts of expressions

Table 4.2: Common Regular Expression Operators and Their Meanings

Examples

1. a—b → Matches either a or b
2. (a—b)*→ Matches any combination (including empty string) of a and b
3. ab* → Matches a, ab, abb, abbb, ...
4. a(a—b)*b→ Matches all strings that start with a and end with b, with any number of a or b in between

4.3 Describing Tokens with REs

Every programming language has a set of tokens which can be described using regular expressions.

1. Keywords

Fixed string patterns like:

```
if | else | while | return | int | float
```

2. Identifiers

Usually follow this pattern:

- Starts with a letter
- Followed by letters or digits

```
[a-zA-Z][a-zA-Z0-9]*
```

Matches:

x, temp, a1, result99

3. Numbers

Integers:

```
[0-9]+
```

Matches: 0, 12, 999

Floating Point:

```
[0-9]+\.[0-9]+
```

Matches: 3.14, 0.0, 100.25

4. Operators Symbols

```
+ | - | * | / | = | == | != | && | || | ( | ) | { | } | ; | ,
```

Each is treated as a literal token in the RE engine. **5. Whitespace Comments**

Whitespace:

```
[\t\n\r ]+
```

Used for skipping spaces

Single-line Comment (C/C++ style):

```
\/\/.*
```

Multi-line Comment:

```
\/\*([^*]|\*+[^*/])*\*+\/
```

4.3.1 Practice – Classify These as Identifiers or Not

Input	Valid Identifier?	Why
`_sum`	No	Starts with _ (depends on rules)
`sum1`	Yes	Valid
`9name`	No	Cannot start with digit
`main()`	No	Includes `()`

Table 4.3: Identifier Validity Examples

4.4 RE to Language

When given a regular expression, you need to:

- Describe the language it generates in words
- Use set-builder notation or precise descriptions
- Give examples of strings in the language and those excluded

Example 1:
RE = a* Step 1: RE a* means zero or more repetitions of a. Step 2: Language: All strings consisting of only as (including the empty string)
Language description:
All strings over a containing zero or more as. This includes the empty string "", "a", "aa", "aaa", and so on.
Example 2:
RE = (a—b)* abb
Step 1: RE (a—b)* abb means: Any string (including empty) over a,b followed by substring "abb"
Step 2: Language description: All strings over a,b that end with the substring "abb"
Example strings in language:

1. "abb"
2. "aabb"
3. "bababb"

Example strings NOT in language:

1. "ab" (does not end with "abb")
2. "aab"
3. "abc" (contains c, not in alphabet)

Example 3:
RE = (ab)*
Step 1: RE (ab)* means zero or more repetitions of "ab".
Step 2: Language: All strings consisting of zero or more repetitions of the substring "ab"
Language description: Strings formed by concatenating "ab" zero or more times, including the empty string.
Example strings in language:

- ""
- "ab"
- "abab"
- "ababab"

Example strings NOT in language:

- "a"
- "b"
- "aba" (ends with an a but not followed by b)
- "abb"

Let's look at some REs and the languages they describe.

Regular Expression	**Language (Set of Strings)**	**Description**
`a*`	{, a, aa, aaa, ...}	Zero or more a's
`ab*`	{a, ab, abb, abbb, ...}	a followed by zero or more b's
`(ab)*`	{, ab, abab, ababab, ...}	Zero or more repetitions of ab
`(a|b)*abb`	All strings ending with `abb`	Matches any a/b pattern ending in abb
`a(a|b)*b`	Strings starting with `a` and ending with `b`	Starts with a, ends with b

Table 4.4: Regular Expressions and Their Languages

Examples:
Match These Strings

```
RE: (a|b)*abb
```

Which of the following are valid?

- abb - YES
- aabb -YES
- ababb -YES
- ababa -NO
- bbab -NO

4.5 Language to RE

When you are given a language described by some property, and asked to find a regular expression (RE) that describes exactly that language, follow these steps:

Step-by-step method: Understand the language description fully Identify all conditions the strings must satisfy. Identify the alphabet involved (e.g., a,b) Break the language into components — prefixes, suffixes, repeated patterns, etc. Express each component as an RE or a combination of REs Combine the parts with union (—), concatenation, and Kleene star (*) operators

Example 1:

Language of strings that start and end with a over a,b

Step 1: Condition: Strings must start with a and end with a.

Step 2: Alphabet: a,b

Step 3: Inside the string (between the first and last a), we can have any combination of a and b (including empty string).

Step 4:

1. Start with a → a
2. Middle part: any string over a,b → (a—b)*
3. End with a → a

Step 5: Combine all: a (a—b)* a

Additional note: The shortest string in the language must be at least length 2 (since it starts and ends with a), except the string "a" itself, which is length 1 and also starts and ends with a.

So include the string "a" separately or handle via (a (a—b)* a) — a

Or simply:

a (a—b)* a — a

which can be written as:

a (a—b)* a — a

Example 2:

Language of strings with even number of bs over a,b

Step 1: Condition: The total number of b characters in the string is even (including zero).

Step 2: Alphabet: a,b

Step 3: Let's reason how to generate strings with even number of bs. 'a' can appear anywhere any number of times — does not affect b count. To keep track of even bs, we need bs in pairs.

Step 4:

a* represents any number of as

b b represents two bs → even number of bs

We can have any sequence of as and pairs of bs in any order.

Step 5: So the building block for even number of bs is:

Zero or more occurrences of: any number of as, followed by two bs, followed by any number of as.

Expressed as:

(a*bba*)*

This RE generates all strings over a,b with an even number of bs.

This means zero or more repetitions of blocks that have pairs of bs separated by any number of as.

4.6 Transition Diagrams

A **transition diagram** is a finite automaton representation of a regular expression. It's a graph:

- Nodes are states
- Edges are transitions labeled with input characters
- One start state, one or more final states

4.6.1 Examples

Example 1:

```
RE = ab*
```

Matches: a, ab, abb, ...

Diagram Description:

```
 State q0 (Start)
   |
   a
   ↓
State q1 (Accepting)
   |
   b (loop)
   ↓
  (q1 again)
```

Example 2

```
RE = (a|b)*abb
```

Matches strings ending in abb
Diagram Outline:

1. From start state q0, loop a or b
2. On seeing a, go to q1
3. On b, go to q2
4. On another b, go to final state q3

4.7 Limitations of Regular Expressions

While REs are powerful, they are not suitable for all language patterns.
What Regular Expressions CAN Do:

- Describe simple tokens and sequences
- Use in lexical analyzers

What Regular Expressions CANNOT Do:

- Match nested structures like balanced parentheses
- Count or compare multiple symbol counts

For those, we use Context-Free Grammars (CFGs).

4.8 Practice Problems

Problem 1: Write an RE for the following tokens a. All strings over a, b where every a is followed by at least one b b. All strings that start with a and end with b c. Valid identifiers starting with letters and followed by digits
Problem 2: Identify Valid Strings For RE:

```
a(a|b)*b
```

Which are accepted?

- ab -YES
- aab -YES
- abb -YES
- ba -NO

- a -NO

Problem 3: Build a Transition Diagram for

```
RE = (a|b)*abb
```

- Create states
- Show transitions
- Label start and accepting state
-

Problem 4:
Write REs for

- Binary numbers divisible by 4 (ending with 00)
- Email IDs (simple version): e.g., name@domain.com
- Floating point constants like: 5.67, 0.1, .99

4.9 Summary

Concept	Explanation	Examples / Notes
Regular Expression (RE)	Formal notation for describing string patterns	Describes token structure in programming languages
Tokens	Smallest units recognized by lexical analyzers	Identifiers, keywords, operators, etc.
Operators	Special symbols used in REs	`|`, `.`, `*`, `+`, `?`
Diagrams	Graphical representation of REs as finite automata	Used in lexical analysis
Limitations	REs can't describe nested structures	E.g., matching parentheses needs CFG

Table 4.5: Key Concepts Related to Regular Expressions

4.10 C Programs Matching RE Patterns

Program 1:
Check if a string is a valid identifier Regex Pattern:

```
[a-zA-Z][a-zA-Z0-9]*
```

CODE:

```
    #include <stdio.h>

int is_letter(char ch) {
    return (ch >= 'A' && ch <= 'Z') || (ch >= 'a' && ch <= 'z');
}

int is_digit(char ch) {
    return ch >= '0' && ch <= '9';
}

int main() {
    char str[100];
    int i = 0, valid = 1;

    printf("Enter the string: ");
    scanf("%s", str);

    if (!is_letter(str[0])) {
        valid = 0;
    } else {
        for (i = 1; str[i] != '\0'; i++) {
            if (!is_letter(str[i]) && !is_digit(str[i])) {
                valid = 0;
                break;
            }
        }
    }

    if (valid)
        printf("Valid Identifier\n");
    else
        printf("Invalid Identifier\n");

    return 0;
}
```

Program 2:

Match a simple integer number Regex Pattern:

```
    [0-9]+
```

CODE:

```
    #include <stdio.h>

int is_digit(char ch) {
```

```
    return ch >= '0' && ch <= '9';
}

int main() {
    char str[100];
    int i = 0, valid = 1;

    printf("Enter the number: ");
    scanf("%s", str);

    while (str[i] != '\0') {
        if (!is_digit(str[i])) {
            valid = 0;
            break;
        }
        i++;
    }

    if (valid)
        printf("Valid Integer\n");
    else
        printf("Invalid Integer\n");

    return 0;
}
```

Program 3:
Match a floating point constant Regex Pattern:

```
[0-9]+\.[0-9]+
```

CODE:

```
#include <stdio.h>

int is_digit(char ch) {
    return ch >= '0' && ch <= '9';
}

int main() {
    char str[100];
    int i = 0, valid = 1, dot_found = 0;

    printf("Enter a float number: ");
    scanf("%s", str);
```

```
    if (str[0] == '.' || str[0] == '\0') {
        valid = 0;
    } else {
        while (str[i] != '\0') {
            if (str[i] == '.') {
                if (dot_found) {
                    valid = 0;
                    break;
                }
                dot_found = 1;
            } else if (!is_digit(str[i])) {
                valid = 0;
                break;
            }
            i++;
        }

        if (!dot_found)
            valid = 0;
    }

    if (valid)
        printf("Valid Float Constant\n");
    else
        printf("Invalid Float Constant\n");

    return 0;
}
```

Program 4:
Match pattern

```
(a|b)abb
```

Accepts any string over a, b that ends in abb
CODE:

```
#include <stdio.h>

int is_valid(char *str) {
    int i = 0;

    while (str[i] != '\0') {
        if (str[i] != 'a' && str[i] != 'b')
            return 0;
```

```
        i++;
    }

    if (i < 3)
        return 0;

    if (str[i - 3] == 'a' && str[i - 2] == 'b' && str[i - 1] == 'b')
        return 1;

    return 0;
}

int main() {
    char str[100];

    printf("Enter string over a, b: ");
    scanf("%s", str);

    if (is_valid(str))
        printf("Valid: Matches (a|b)*abb\n");
    else
        printf("Invalid: Does NOT match (a|b)*abb\n");

    return 0;
}
```

Program 5:

Match keyword from a set Match if input is a keyword like **if, else, while, int**

CODE:

```
#include <stdio.h>

int is_match(char *s, char *t) {
    int i = 0;
    while (s[i] != '\0' && t[i] != '\0') {
        if (s[i] != t[i])
            return 0;
        i++;
    }
    return s[i] == '\0' && t[i] == '\0';
}

int main() {
    char str[100];
```

```
    char *keywords[] = {"if", "else", "while", "int", "return"};
    int found = 0, i;

    printf("Enter a word: ");
    scanf("%s", str);

    for (i = 0; i < 5; i++) {
        if (is_match(str, keywords[i])) {
            found = 1;
            break;
        }
    }

    if (found)
        printf("Valid Keyword\n");
    else
        printf("Not a Keyword\n");

    return 0;
}
```

Chapter 5

Finite Automata – NFA and DFA

5.1 Introduction to Finite Automata

Finite Automata are abstract mathematical models that are used to describe and recognize regular languages — the simplest types of languages accepted by machines. In compiler design, Finite Automata (FA) are the backbone of lexical analysis, where they help identify meaningful units in the source code, called tokens.

A token could be:

- A keyword like if, else, while
- An identifier like sum, counter1
- An integer like 12, 4300
- A symbol like , , +, ==

Why We Need Finite Automata

Regular expressions describe patterns, but finite automata are used to implement them. When we say a token matches a pattern, it means the finite automaton accepted the input string by transitioning through states.

The output of regular expression → finite automaton → lexer is what actually powers compilers.

5.2 Types of Finite Automata

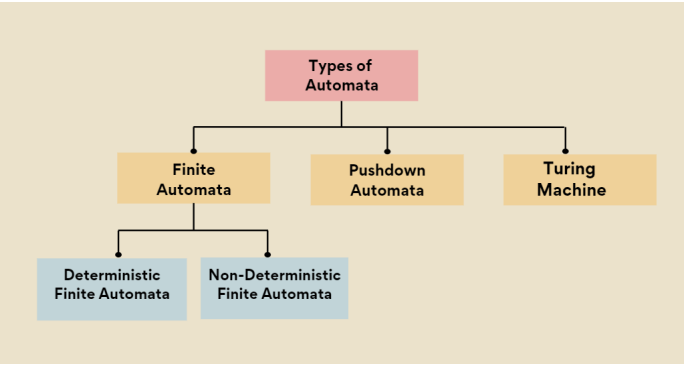
There are mainly two types:

1. Deterministic Finite Automaton (DFA)

- For each state and input symbol, only one transition exists.
- There are no -transitions.
- Every string in the language has only one path through the machine.
- Faster and easier to implement in real compilers.

2. Nondeterministic Finite Automaton (NFA)

- Allows multiple transitions for the same input symbol from a single state.
- Can include -transitions (transitions without consuming input).
- Simpler to design but more abstract.
- Needs to be converted to DFA for real implementation.



5.3 Formal Definition of Finite Automaton

A Finite Automaton (DFA or NFA) is defined by a 5-tuple:

$$\text{FA} = (Q, \Sigma, \delta, q_0, F)$$

Where:

Symbol	Meaning
Q	Finite set of states
Σ	Input alphabet (finite set of symbols)
δ	Transition function: $\delta : Q \times \Sigma \to Q$
q_0	Start state, where $q_0 \in Q$
F	Set of final (accepting) states, $F \subseteq Q$

Table 5.1: Components of a Finite Automaton (FA)

5.4 DFA Concepts & Diagrams

How DFA Works:

1. Start at q
2. For each input symbol, exactly one path is followed
3. If at the end of the input, the machine ends in an accepting state, the string is accepted

Example 1:
DFA for strings over 0,1 that end with 01
Language L = w 0,1* — w ends with 01

State	Meaning
q_0	Start state
q_1	Last symbol was `0`
q_2	Last two symbols were `01`

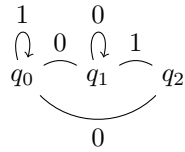
Table 5.2: Transition Table of the Finite Automaton

Transition Table

State	Input 0	Input 1
q_0	q_1	q_0
q_1	q_1	q_2
q_2	q_1	q_0

Table 5.3: States and Meanings in FA

Transition Diagram



Accepted Strings:

- 01
- 101
- 00001

Rejected Strings:

- 0, 11, 0011

5.5 NFA Concepts and Diagrams

How NFA Works:

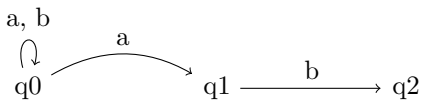
- For a given state and symbol, more than one next state may exist.
- May also move on (epsilon) transitions — without consuming input.
- Accepts a string if any one path through the machine ends in an accepting state.

Example 2:
NFA that accepts strings over a, b ending with ab
States: q0 (start), q1, q2 (accepting)

From	Input	To
q0	a	q0
q0	b	q0
q0	a	q1
q1	b	q2

Table 5.4: Transition Table

Diagram:



Accepts:
ab, aab, bab, aaab
Rejects:
a, b, ba

5.6 Conversion from NFA to DFA

What is an NFA?

- NFA (Nondeterministic Finite Automaton): At any state, for a given input symbol, the NFA can move to any number of next states (including zero).
- It can have multiple possible next states for the same input from one state.

What is a DFA?

DFA (Deterministic Finite Automaton): For any state and input symbol, there is exactly one next state.

Why Convert NFA → DFA?

- DFA is easier to implement as it has no ambiguity in moves.
- Both NFA and DFA recognize the same class of languages (regular languages).
- Conversion uses subset construction.

Algorithm: Subset Construction

1. Start with the start state of the NFA, denoted as q_0.
2. The start state of the DFA is the set $\{q_0\}$.
3. Each DFA state corresponds to a subset of NFA states.
4. For each DFA state S (a subset of NFA states) and for each input symbol a:
   - Compute the set of all possible NFA states reachable from any state in S on input a.
   - This set is the next DFA state for input a.
5. Mark the DFA state as accepting if it contains any accepting NFA state.
6. Repeat the above steps until no new DFA states are added.

Example 1:
Given NFA:

State	**Input** a	**Input** b	**Accepting?**
q_0	$\{q_0, q_1\}$		No
q_1		$\{q_2\}$	No
q_2			Yes

Table 5.5: DFA Transition Table with Accepting States

Explanation:

- From q0 on input a, NFA can stay in q0 or go to q1.
- From q1 on input b, it goes to q2.
- q2 is an accepting state.

DFA State	Input a	Input b	Accepting?
$\{q_0\}$	Move from q_0 on a: $\{q_0, q_1\}$	Move from q_0 on b: $\emptyset$	No
$\{q_0, q_1\}$	From q_0 on a: $\{q_0, q_1\}$, from q_1 on a: $\emptyset \rightarrow$ union: $\{q_0, q_1\}$	From q_0 on b: $\emptyset$, from q_1 on b: $\{q_2\} \rightarrow \{q_2\}$	Yes (contains q_2)
$\emptyset$	$\emptyset$	$\emptyset$	No
$\{q_2\}$	From q_2 on a: $\emptyset$	From q_2 on b: $\emptyset$	Yes

Table 5.6: DFA transition table with input moves and accepting states

Step 1: Start state of DFA = $\{q_0\}$.
We'll denote DFA states as sets of NFA states, e.g., $\{q_0, q_1\}$. **Step 2:** Compute transitions for DFA states:

- **States:**
$$\{\{q_0\}, \{q_0, q_1\}, \emptyset, \{q_2\}\}$$
- **Start State:**
$$\{q_0\}$$
- **Accepting States:**
$$\{q_0, q_1\} \quad \text{(since it contains } q_2\text{)}, \quad \{q_2\}$$

DFA Transition Table:

DFA State	Input a	Input b
$\{q_0\}$	$\{q_0, q_1\}$	$\emptyset$
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_2\}$
$\emptyset$	$\emptyset$	$\emptyset$
$\{q_2\}$	$\emptyset$	$\emptyset$

Table 5.7: DFA Transition Table

5.7 DFA vs NFA: In-Depth Comparison

Feature	DFA	NFA
Transitions	Exactly one per input symbol	Can have multiple per symbol
ε-transitions	Not allowed	Allowed
Design	More complex	Easier
Execution	Straightforward	Requires simulation
Used in Lexical Analyzer	Yes	Only after NFA $\rightarrow$ DFA conversion

Table 5.8: Comparison between DFA and NFA

5.8 Practice Problems

1.Given an NFA:

State	Input 0	Input 1	Accepting?
s0	{s1}	{s0, s2}	No
s1	{s2}		No
s2		{s1}	Yes

Table 5.9: NFA State Transitions and Acceptance

Convert to DFA.
2.Given an NFA:

State	Input a	Input b	Accepting?
q0	{q1}	{q0}	No
q1	{q2}		Yes
q2		{q1}	No

Table 5.10: NFA State Transitions and Acceptance

Convert to DFA.

5.9 Summary

Concept	Summary
FA	Mathematical model to accept/reject strings
DFA	One transition per input, no ε
NFA	Multiple transitions, with/without ε
ε-NFA	Free moves via ε
NFA $\rightarrow$ DFA	Subset construction method
Application in Compiler	Lexical Analyzer (Token recognition)

Table 5.11: Summary of Finite Automata Concepts

5.10 C programming Problems

Problem 1: DFA for String Ending in 01

C Code

```
#include <stdio.h>
#include <string.h>
int main() {
```

```
    char str[100]; int state=0,i;
    printf("Enter binary string: ");
    scanf("%s", str);
    for(i=0; str[i]!='\0'; i++) {
        if(state==0 && str[i]=='0') state=1;
        else if(state==0 && str[i]=='1') state=0;
        else if(state==1 && str[i]=='0') state=1;
        else if(state==1 && str[i]=='1') state=2;
        else if(state==2 && str[i]=='0') state=1;
        else if(state==2 && str[i]=='1') state=0;
    }
    if(state==2) printf("Accepted\n");
    else printf("Rejected\n");
    return 0;
}
```

Input/Output

```
Input: 1101
Output: Accepted
```

Problem 2: NFA Simulation for a*b

C Code

```
#include <stdio.h>
int main() {
    char str[100]; int i=0;
    printf("Enter string: ");
    scanf("%s", str);
    while(str[i]=='a') i++;
    if(str[i]=='b' && str[i+1]=='\0') printf("NFA Accepted\n");
    else printf("Rejected\n");
    return 0;
}
```

Input/Output

```
Input: aaab
Output: NFA Accepted
```

Problem 3: Count Number of a's in DFA

C Code

```
#include <stdio.h>
#include <string.h>
int main() {
    char str[100]; int count=0,i;
    printf("Enter string: ");
    scanf("%s", str);
    for(i=0;str[i]!='\0';i++)
        if(str[i]=='a') count++;
    printf("Number of a's: %d\n", count);
    return 0;
}
```

Input/Output

```
Input: aabaa
Output: Number of a's: 4
```

Chapter 6

Left Recursion & Left Factoring

In compiler design, especially during syntax analysis (parsing), grammars must be designed in a way that parsers can interpret efficiently. However, left recursion and grammar ambiguity create serious issues for top-down parsers like recursive descent or LL(1) parsers.

This chapter discusses:

- What left recursion is and why it must be eliminated
- What left factoring is and why it is necessary
- How to transform grammars using formal steps
- Detailed examples and practice exercises

6.1 What Is Left Recursion?

A grammar is left recursive if a non-terminal symbol calls itself as the first symbol on the right-hand side (RHS) of its production.

Formal Definition:

A production is left-recursive if it has the form: A production with left recursion:

$$A \to A\alpha \mid \beta$$

Where:

- A is a non-terminal
- α is a non-empty string of grammar symbols
- β is any production that does not begin with A

To eliminate left recursion, we rewrite it as:

$$A \to \beta A'$$
$$A' \to \alpha A' \mid \varepsilon$$

Where:

- A' is a new non-terminal
- ε is the empty string

Why Left Recursion is a Problem:
Top-down parsers cannot handle left recursion as it causes infinite recursion (non-termination).
Example of failure:

```
A → A + a | b
```

Trying to parse b+a leads to infinite expansion: A → A + a → A + a + a → ...

6.2 Eliminating Immediate Left Recursion

Rule:
If grammar has:

$$A \to A\alpha_1 \mid A\alpha_2 \mid \ldots \mid \beta_1 \mid \beta_2 \mid \ldots$$

Where:

- A is a non-terminal
- $\alpha_1, \alpha_2, \ldots$ are non-empty sequences of grammar symbols
- $\beta_1, \beta_2, \ldots$ are alternatives that do not begin with A

Then replace with:

$$A \to \beta_1 A' \mid \beta_2 A' \mid \ldots$$
$$A' \to \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \varepsilon$$

Example 1:
Original:

```
A → A + a | b
```

Step-by-step:

- Left-recursive part: A + a

- Non-recursive part: b

Transformed:

$$A \to b\ A'$$
$$A' \to +\ a\ A' \mid \varepsilon$$

Example 2:
Original:

```
Expr → Expr + Term | Term
```

Rewritten:

$$\text{Expr} \to \text{Term Expr}'$$
$$\text{Expr}' \to +\ \text{Term Expr}' \mid \varepsilon$$

6.3 Eliminating Indirect Left Recursion

Indirect left recursion occurs when a non-terminal leads back to itself through one or more other non-terminals.

Example 3:

$$A \to B\,\alpha$$
$$B \to A\,\beta \mid \gamma$$

This is indirect because A → B → A.
Steps to Remove Indirect Left Recursion:

1. Order all non-terminals as A, A, ..., An
2. For each Ai:

- Replace any production Ai → Aj where j ¡ i
- Then eliminate immediate left recursion on Ai

Example 4:
Given:

```
S → A a | b
A → S b | c
```

Rewriting order: A = S, A = A
Substitute S into A:

```
A → A a b | b b | c
```

Now eliminate immediate left recursion:

$$A \to\ bb\ A' \ \mid\ c\ A'$$
$$A' \to\ ab\ A' \ \mid\ \varepsilon$$

6.4 Introduction to Left Factoring

Sometimes multiple productions for a non-terminal begin with the same prefix, making it hard for a parser to decide which production to use.

Problem:

```
Stmt → if E then S else S
 | if E then S
```

Here, both start with if E then S.

A predictive parser (like LL(1)) will not know whether to take the first or second rule without looking ahead.

Left Factoring Rule:

Factor out common prefixes:

$$A \to \alpha A'$$
$$A' \to \beta_1 \mid \beta_2$$

Example 5:

Original:

```
A → a b c | a b d
```

Factored:

$$A \to a\, b\, A'$$
$$A' \to c \mid d$$

Now it's suitable for LL(1) parsing.

Example 6: (Programming Language Syntax)

Original:

```
Stmt → if E then S else S
 | if E then S
```

Factored:

$$\text{Stmt} \to \text{if } E \text{ then } S \text{ Stmt'}$$
$$\text{Stmt'} \to \text{else } S \mid \varepsilon$$

6.5 Left Recursion vs Left Factoring

Feature	Left Recursion	Left Factoring
Problem Solved	Infinite recursion	Lookahead confusion
Affects	Left-recursive grammars	Grammars with shared prefixes
Solved By	Introducing new non-terminals and rearranging	Factoring common prefixes into new productions
Target Parser	Top-down	LL(1), Recursive descent

Table 6.1: Comparison between Left Recursion and Left Factoring

6.6 Practice Problems

Problem 1: Remove Left Recursion

$$\begin{cases} A \to A + B \mid B \\ B \to \text{id} \end{cases}$$

Answer:

$$\begin{cases} A \to BA' \\ A' \to +BA' \mid \varepsilon \end{cases}$$

Problem 2: Left Factoring

$$A \to xyz \mid xyw$$

Answer:

$$\begin{cases} A \to xyA' \\ A' \to z \mid w \end{cases}$$

Problem 3: Detect and Eliminate Indirect Left Recursion

$$\begin{cases} A \to Ba \mid c \\ B \to Ab \mid d \end{cases}$$

Problem 4: Factor

$$S \to \text{while } E \text{ do } S \mid \text{while } E \text{ do } S \text{ else } S$$

Answer:

$$S \to \text{while } E \text{ do } S\ S'$$
$$S' \to \text{else } S \mid \varepsilon$$

6.7 Summary

Concept	Purpose
Left Recursion	Must be eliminated for top-down parsers
Immediate	$A \to A\alpha$
Indirect	$A \to B \to A$
Left Factoring	Needed when productions share prefixes
LL(1) Grammar	Requires both transformations

Table 6.2: Summary of Left Recursion and Left Factoring Concepts

6.8 C programming Problems

Problem 1: Eliminate Left Recursion (Simple Simulation)

C Code

```
#include <stdio.h>
int main() {
    printf("Grammar: A->Aa|b\n");
    printf("After removing left recursion:\n");
    printf("A->bR\n");
    printf("R->aR|\n");
    return 0;
}
```

Output

```
Grammar: A->Aa|b
After removing left recursion:
A->bR
R->aR|
```

Problem 2: Left Factoring Example

C Code

```
#include <stdio.h>
int main() {
    printf("Grammar: A->ab|ac\n");
    printf("After Left Factoring:\n");
    printf("A->aB\n");
```

```
    printf("B->b|c\n");
    return 0;
}
```

Problem 3: Parse Given String Using Transformed Grammar

C Code

```
#include <stdio.h>
int main() {
    char str[10];
    printf("Enter string (ab/ac): ");
    scanf("%s", str);
    if(str[0]=='a' && (str[1]=='b'||str[1]=='c') && str[2]=='\0')
        printf("Accepted\n");
    else
        printf("Rejected\n");
    return 0;
}
```

Input/Output

```
Input: ac
Output: Accepted
```

Chapter 7

FIRST and FOLLOW Sets

7.1 Introduction: Why FIRST and FOLLOW Matter

When building a parser—especially a predictive (top-down) parser like LL(1)—we must determine which grammar rule to apply by looking at one token at a time. This requires predictive power. That's where **FIRST** and **FOLLOW** sets help.

Why These Sets Are Crucial

- **FIRST(X)** helps predict what symbols could start a derivation from X.
- **FOLLOW(X)** tells us what symbols may follow X in any derivation.

Without these, a top-down parser would:

- Guess rules blindly, or
- Require backtracking, which is inefficient and difficult to implement.

For LL(1) parsing, each cell in the parsing table must contain only one production. This is only possible if:

- **FIRST** and **FOLLOW** are calculated correctly, and
- The grammar is **left-recursion-free** and **left-factored**.

7.2 FIRST Set – Definition and Purpose

What Is the FIRST Set?

The **FIRST** set of a grammar symbol (terminal or non-terminal) contains the terminals that can appear at the beginning of strings derived from that symbol.

If $A \Rightarrow^* a\alpha$, then $a \in \text{FIRST}(A)$.
It also includes ε if the symbol can derive an empty string.

Computation Rules for FIRST(X)

Let X be any grammar symbol.

Rule	Description
R1	If X is a terminal, then $\text{FIRST}(X) = \{X\}$
R2	If $X \to \varepsilon$, then $\varepsilon \in \text{FIRST}(X)$
R3	If $X \to Y_1Y_2Y_3 \ldots$, then add to $\text{FIRST}(X)$ all symbols from $\text{FIRST}(Y_1)$
R4	If $\varepsilon \in \text{FIRST}(Y_1)$, then add $\text{FIRST}(Y_2)$, and so on
R5	If all Y_i can derive ε, then $\varepsilon \in \text{FIRST}(X)$

Table 7.1: Rules for Computing FIRST Sets

Examples

Example 1: Simple Terminal-Only Grammar

Grammar:

$$S \to a$$

Explanation: S derives the terminal a, so:

$$\text{FIRST}(S) = \{a\}$$

Example 2: Nullable Non-terminal

Grammar:

$$A \to \varepsilon \mid a$$

Explanation: Two productions: one derives ε, and the other derives a, so:

$$\text{FIRST}(A) = \{\varepsilon, a\}$$

Example 3: Sequence Derivations

Grammar:

$$S \to A\ BA \to \varepsilon \mid aB \to b$$

Step-by-step:

- FIRST(A) = { a, ε }
- Since A is nullable, move to B: FIRST(B) = { b }
- Therefore, FIRST(S) = FIRST(A) - { ε } FIRST(B) = { a, b }

Example 4: Recursive Derivation

Original Grammar:

$$E \to E + T \mid TT \to id$$

Left Recursion Removed:

$$E \to T\ E'E' \to +T\ E' \mid \varepsilon T \to id$$

Now compute:

$$\begin{aligned}\text{FIRST}(T) &= \{id\}\\ \text{FIRST}(E) &= \text{FIRST}(T) = \{id\}\\ \text{FIRST}(E') &= \{+, \varepsilon\}\end{aligned}$$

Common Pitfalls

- Don't forget to check each symbol in sequence if previous ones are nullable.
- Always compute `FIRST` recursively for non-terminals.
- Only include ε in `FIRST(X)` if all symbols in a production can derive ε.

7.3 FOLLOW Set - Definition and Purpose

What Is the FOLLOW Set?

The **FOLLOW** set of a non-terminal A is the set of terminals that can appear immediately after A in some derivation.

In other words, if there is a derivation $S \Rightarrow^* \alpha A a \beta$, then $a \in \text{FOLLOW}(A)$.

If A is the start symbol, then the end-of-input marker \$ is also in $\text{FOLLOW}(A)$.

Computation Rules for FOLLOW(A)

Let A, B, C be non-terminals, and α, β be any grammar symbols.

Rule	Description
R1	If S is the start symbol, then \$ $\in$ FOLLOW(S).
R2	If there is a production $A \to \alpha B \beta$, then FIRST(β) $-\{\varepsilon\} \subseteq$ FOLLOW(B).
R3	If $A \to \alpha B$ or $A \to \alpha B \beta$ and $\beta \Rightarrow^* \varepsilon$, then FOLLOW($A$) $\subseteq$ FOLLOW(B).

Table 7.2: Rules for Computing FOLLOW Sets

Examples

Example 1: Basic FOLLOW Sets

Grammar:

$$S \to A\ BA \to a \mid \varepsilon B \to b$$

Start Symbol: $S \Rightarrow \$ \in \text{FOLLOW}(S)$
Now compute:

- $\text{FOLLOW}(A)$: Since A is followed by B, and $\text{FIRST}(B) = \{b\}$, so $\text{FOLLOW}(A) = \{b\}$
- $\text{FOLLOW}(B) = \text{FOLLOW}(S) = \{\$\}$

Example 2: Complex FOLLOW Computation

Grammar:

$$S \to A\ B\ CA \to \varepsilon B \to \varepsilon C \to c$$

Start Symbol: $\$ \in \text{FOLLOW}(S)$
Step-by-step:

- A is followed by B, and $\text{FIRST}(B) = \{\varepsilon\}$, so continue to C
- $\text{FIRST}(C) = \{c\}$, so $\text{FOLLOW}(A) = \text{FOLLOW}(B) = \{c\}$
- $\text{FOLLOW}(C) = \text{FOLLOW}(S) = \{\$\}$

Example 3: FOLLOW with Recursion

Grammar:

$$E \to T\ E'E' \to +\ T\ E' \mid \varepsilon T \to id$$

Start Symbol: $\$ \in \text{FOLLOW}(E)$
Step-by-step:

- E' follows $E \Rightarrow \text{FOLLOW}(E) \subseteq \text{FOLLOW}(E') = \{\$\}$
- T is followed by E'. So $\text{FOLLOW}(T) \supseteq \text{FIRST}(E') - \{\varepsilon\} = \{+\}$
- Also, since $E' \Rightarrow \varepsilon$, $\text{FOLLOW}(E) \subseteq \text{FOLLOW}(T) \Rightarrow \$ \in \text{FOLLOW}(T)$
- Final: $\text{FOLLOW}(T) = \{+, \$\}$

Special Case: When FIRST() Has

If $A \to \alpha B \beta$ and $\beta \Rightarrow^* \varepsilon$, then $\text{FOLLOW}(A) \subseteq \text{FOLLOW}(B)$

This ensures that FOLLOW information flows when the remaining symbols after B can be nullable.

7.4 Combined Table for FIRST and FOLLOW

Non-terminal	FIRST	FOLLOW
S	$\{a, b\}$	$\{\$\}$
A	$\{a, \varepsilon\}$	$\{b, \$\}$
B	$\{b, \varepsilon\}$	$\{\$\}$

Table 7.3: FIRST and FOLLOW sets

7.5 How FIRST and FOLLOW Are Used in LL(1) Parsers

Constructing the LL(1) Parse Table

In LL(1) parsing, we build a predictive parsing table using the **FIRST** and **FOLLOW** sets.

Parse Table Dimensions

The table has:

- Rows = Non-terminals
- Columns = Terminals (including \$ for end of input)

Filling Rules

For each production $A \to \alpha$, do the following:

1. For each terminal $a \in \text{FIRST}(\alpha)$, place $A \to \alpha$ in cell $[A, a]$.
2. If $\varepsilon \in \text{FIRST}(\alpha)$, then for each terminal $b \in \text{FOLLOW}(A)$, place $A \to \alpha$ in cell $[A, b]$.

Example: Constructing a Predictive Parse Table

Grammar:

$$S \to A\ BA \to a \mid \varepsilon B \to b$$

FIRST Sets

$$\begin{aligned}
&\text{FIRST}(A) = \{a, \varepsilon\} \\
&\text{FIRST}(B) = \{b\} \\
&\text{FIRST}(S) = \text{FIRST}(A) - \{\varepsilon\} \cup \text{FIRST}(B) = \{a, b\}
\end{aligned}$$

FOLLOW Sets

$$\text{FOLLOW}(S) = \{\$\}$$
$$\text{FOLLOW}(A) = \text{FIRST}(B) = \{b\}$$
$$\text{FOLLOW}(B) = \text{FOLLOW}(S) = \{\$\}$$

LL(1) Parse Table

Non-Terminal	a	b	$
S	$S \to A\ B$	$S \to A\ B$	
A	$A \to a$	$A \to \varepsilon$	
B		$B \to b$	

Table 7.4: LL(1) Parse Table for the Grammar

Explanation

- For $S \to AB$: FIRST(A) includes a, ε. Since $\varepsilon \in \text{FIRST}(A)$, also look at FIRST(B) = $\{b\}$. So add the rule to columns a and b.
- For $A \to a$: add to column a
- For $A \to \varepsilon$: since $\varepsilon \in \text{FIRST}(A)$, also add to column b because $b \in \text{FOLLOW}(A)$
- For $B \to b$: put in column b

7.6 Special Cases

in FIRST set: Only include ε in FIRST(A) if $A \to \varepsilon$ is a production.
Use **FOLLOW** sets when deciding which ε-rule to apply.

Left Recursion: **FIRST** and **FOLLOW** sets help detect if the grammar needs transformation.

Left recursion must be removed before using LL(1) parsers.

7.7 LL(1) Grammar Validity Test Using FIRST/FOLLOW

A grammar is said to be **LL(1)** if it can be parsed using:

- Left-to-right scanning of the input (the first **L**)
- Leftmost derivation (the second **L**)

- Using **1** lookahead symbol

To ensure that a grammar is LL(1), the following conditions must hold for every non-terminal A, and for any two different productions:

$$A \to \alpha \quad \text{and} \quad A \to \beta$$

1. **FIRST sets must be disjoint:**

$$\text{FIRST}(\alpha) \cap \text{FIRST}(\beta) = \emptyset$$

2. **If** $\varepsilon \in \text{FIRST}(\alpha)$, **then:**

$$\text{FIRST}(\beta) \cap \text{FOLLOW}(A) = \emptyset$$

3. **If** $\varepsilon \in \text{FIRST}(\beta)$, **then:**

$$\text{FIRST}(\alpha) \cap \text{FOLLOW}(A) = \emptyset$$

Purpose

These conditions ensure that:

- The parser can decide which production to apply with a single lookahead.
- There is no ambiguity or need for backtracking.

7.8 Practice Exercises

Problem 1: Grammar:

$$S \to ABA \to a \mid \varepsilon B \to b \mid \varepsilon$$

Compute the **FIRST** and **FOLLOW** sets for S, A, and B.

Problem 2: Grammar:

$$E \to TE' E' \to +TE' \mid \varepsilon T \to \text{id}$$

Find:

$$\text{FIRST}(E), \quad \text{FIRST}(E'), \quad \text{FIRST}(T)$$

$$\text{FOLLOW}(E), \quad \text{FOLLOW}(E'), \quad \text{FOLLOW}(T)$$

Problem 3: Build the LL(1) parse table using **FIRST** and **FOLLOW** sets for the grammar:

$$S \to ABA \to a \mid \varepsilon B \to b$$

7.9 Summary Table

Concept	Description
FIRST(X)	Terminals that begin strings derivable from X
FOLLOW(X)	Terminals that appear immediately after X
ε Inclusion	Used only when a production can go completely empty
Use in Parser	FIRST $\to$ Choose production, FOLLOW $\to$ when to apply ε
LL(1) Table	Built using FIRST and FOLLOW sets to avoid backtracking

7.10 C programming Problems

Problem 1: Compute FIRST for 2 Productions

C Code:

```
#include <stdio.h>
int main() {
    printf("Grammar:\nE->TA\nA->+TA|\n");
    printf("FIRST(E) = { (,id }\n");
    printf("FIRST(A) = { +, }\n");
    return 0;
}
```

Explanation:

- `E` produces `T A`. Since `T` can start with `(` or `id`, these are in `FIRST(E)`.
- `A` produces either `+ T A` or ϵ. So `FIRST(A)` = { +, ϵ }.

Problem 2: Compute FOLLOW for 2 Productions

C Code:

```
#include <stdio.h>
int main() {
    printf("Grammar:\nE->TA\nA->+TA|\n");
    printf("FOLLOW(E) = { $, ) }\n");
    printf("FOLLOW(A) = { $, ) }\n");
    return 0;
}
```

Explanation:

- `E` is the start symbol, so $ is in `FOLLOW(E)`. Also, from the grammar context, closing parenthesis `)` follows `E`.
- `A` follows `T` in `E->TA`, so whatever follows `E` (which is $,)) also follows `A`.

Problem 3: Parse String `id+id` Using FIRST & FOLLOW

C Code:

```
#include <stdio.h>
int main() {
    printf("String: id+id\n");
    printf("Parsing using FIRST/FOLLOW Table:\n");
    printf("Step1: E => TA\n");
    printf("Step2: T => id\n");
    printf("Step3: A => +TA\n");
    printf("Step4: T => id\n");
    printf("Accepted\n");
    return 0;
}
```

Explanation:

- Start with $E \Rightarrow TA$.
- T matches the first `id`.
- A matches `+ T A` since the next token is `+`.
- Next T matches the second `id`.
- Parsing completes successfully and the string is accepted.

Chapter 8

LR(0) Parser and Canonical Table

8.1 Introduction to Bottom-Up Parsing

In the earlier chapters, we discussed how **top-down parsers** like **LL(1)** predict which production to apply using **FIRST** and **FOLLOW** sets. These parsers work by expanding the parse tree from the *top* (start symbol) to the *leaves*.

However, there are limitations:

- LL(1) parsers **cannot handle left recursion**.
- LL(1) grammars are **restrictive** and must be **left-factored**.

Bottom-up parsers, in contrast, start from the *input tokens* and work their way *back up* to the start symbol by **reducing** the input string step by step.

Among bottom-up parsers, the **LR parsers** (Left-to-right scan, Rightmost derivation in reverse) are the most powerful and widely used in compiler design.

Why Study LR(0) Parsers?

- LR(0) forms the **foundation** for more powerful LR parser types, such as:
  - **SLR (Simple LR)**
  - **LALR (Look-Ahead LR)**
  - **Canonical LR**
- It is used in **automatic parser generation tools** like `YACC` and `Bison`.

- It helps you understand the **structure** of parsing tables and **detect ambiguity**.
- It trains you to **think like a compiler** by recognizing derivation patterns and decision points.

8.2 What is an LR(0) Item?

An **LR(0) item** is a grammar production with a dot (•) indicating how much of the production has been recognized so far during parsing.

For example, consider the production:

$$E \to E + T$$

The corresponding LR(0) items are:

1. $E \to \bullet E + T$ (Nothing matched yet)
2. $E \to E \bullet + T$ (The non-terminal E has been recognized)
3. $E \to E + \bullet T$ ('+' symbol seen after E)
4. $E \to E + T \bullet$ (All symbols matched, ready to reduce)

Each **state** in an LR(0) parser corresponds to a set of such items.

8.3 Augmented Grammar – Why and How

To design an LR(0) parser, we begin by **augmenting the grammar**. This augmentation explicitly defines when parsing should stop.

Original Grammar

$$S \to aSb \mid \varepsilon$$

Augmented Grammar

$$S' \to S$$

$$S \to aSb \mid \varepsilon$$

The production $S' \to S$ helps the parser detect when the original start symbol S is completely parsed and ready to be accepted.

8.4 Constructing LR(0) Items and States (With Examples)

Grammar:

$$\begin{cases} S' \to S \\ S \to a\,S\,b \mid \varepsilon \end{cases}$$

Step 1: Start Item and Closure

Start with the item:

$$I_0: \quad S' \to \bullet S$$

Compute the closure of I_0:

$$S \to a\,S\,b \quad \Rightarrow \quad S \to \bullet a\,S\,b$$

$$S \to \varepsilon \quad \Rightarrow \quad S \to \bullet \varepsilon$$

So the full I_0 is:

$$\begin{cases} S' \to \bullet S \\ S \to \bullet a\,S\,b \\ S \to \bullet \varepsilon \end{cases}$$

Step 2: GOTO from I_0

Apply GOTO(I_0, S): shift dot over S in $S' \to \bullet S$

$$I_1: \quad S' \to S\bullet$$

Apply GOTO(I_0, a): shift dot over a in $S \to \bullet aSb$

$$I_2: \quad \begin{cases} S \to a \bullet Sb \\ S \to \bullet aSb \\ S \to \bullet \varepsilon \end{cases}$$

Note: Closure on I_2 adds the last two items since after the dot is S, a non-terminal.

Continue Computing GOTO

$$I_3 = \text{GOTO}(I_2, S): \quad S \to aS \bullet b$$

$$I_4 = \text{GOTO}(I_3, b): \quad S \to aSb\bullet$$

$$I_5 = \text{GOTO}(I_0, \varepsilon): \quad S \to \varepsilon\bullet$$

Canonical Collection Overview

State	Items
I_0	$S' \to \bullet S$ $S \to \bullet aSb$ $S \to \bullet \varepsilon$
I_1	$S' \to S\bullet$
I_2	$S \to a \bullet Sb$ $S \to \bullet aSb$ $S \to \bullet \varepsilon$
I_3	$S \to aS \bullet b$
I_4	$S \to aSb\bullet$
I_5	$S \to \varepsilon\bullet$

Table 8.1: Canonical Collection of LR(0) Items

Each item set corresponds to a state in the LR(0) parser.

Next steps would include constructing the **ACTION** and **GOTO** tables using these states for the full LR(0) parsing automaton.

8.5 Parsing Table (ACTION, GOTO)

Terminals: $\{a, b, \$\}$ Non-terminals: $\{S\}$

State	a	b	$	S
0	$s2$			1
1			acc	
2	$s2$			3
3		$s4$		
4			r_1	
5			r_2	

Table 8.2: LR(0) Parsing Table

Legend:

- sX: shift and go to state X
- rY: reduce using production Y
- acc: accept the input

Productions:

$$\begin{cases} r_1 : S \to a\, S\, b \\ r_2 : S \to \varepsilon \end{cases}$$

8.6 String Parsing Example (Step-by-Step)

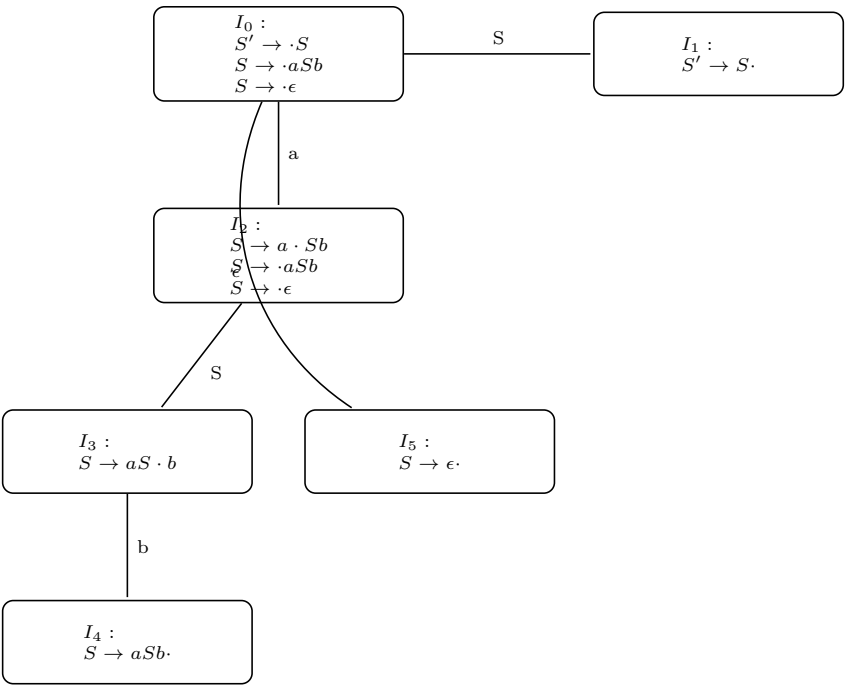
Initial:
Stack: \$ 0
Input: $a\, a\, b\, b\, \$$

Step	Stack	Input	Action
1	\$0	$aabb\$$	$s2$
2	\$0$a2$	$abb\$$	$s2$
3	\$0$a2a2$	$bb\$$	r_2
4	\$0$a2a2S3$	$bb\$$	—
5	\$0$a2a2S3b4$	$b\$$	$s4$
6	\$0$a2a2S3b4$	$b\$$	r_1
7	\$0$a2S3$	$b\$$	—
8	\$0$a2S3b4$	\$	$s4$
9	\$0$a2S3b4$	\$	r_1
10	\$0$S1$	\$	—
11	\$0$S1$	\$	acc

Conclusion: The input string `a a b b` is accepted by the parser.

8.7 Sketching LR(0) State Transition Diagram

tikz



8.8 Conflict Detection in LR(0)

LR(0) parsers may face conflicts when:

- A state contains both:

  $$\text{A completed item: } A \to \alpha\cdot$$

  and

  $$\text{An item with a dot before a symbol: } B \to \cdot a\beta$$

  This situation creates a **shift/reduce conflict**.

- A state contains two or more completed items:

  $$A \to \alpha\cdot \quad \text{and} \quad B \to \beta\cdot$$

  This causes a **reduce/reduce conflict**.

Consider the grammar:

$$S \to A\,a \mid B\,a$$
$$A \to c$$
$$B \to c$$

Problem: Both A and B start with the terminal c.

During parsing, upon seeing the input symbol c, the parser cannot decide whether to reduce to A or B.

This ambiguity causes a conflict, making the grammar **not LR(0)**.

8.9 LR(0) vs Other Parsers

Parser	Lookahead	Power	Examples
LL(1)	1 symbol	Less powerful	Top-down
LR(0)	0	More powerful	Bottom-up
SLR	1 + FOLLOW	Stronger	Uses FOLLOW
LALR	1 + merge	Used in real tools like YACC	

Table 8.3: Comparison of parser types

8.10 Practice Exercises

Exercise 1: Basic LR(0) Parsing Construction

Given Grammar: [language=] S → A B A → a B → b

Tasks:

1. Augment the grammar.
2. Construct all LR(0) items.
3. Build the canonical collection of LR(0) item sets.
4. Construct the ACTION and GOTO parsing table.

Exercise 2: LR(0) Parsing Example with Input

Grammar: [language=] S → a S b —

Input: `a a b b`

Tasks:

1. Draw all LR(0) item sets for this grammar.
2. Build the complete LR(0) parsing table.
3. Show complete step-by-step parsing process for the input string.

Exercise 3: LR(0) Conflict Example

Grammar: [language=] E → E + E — id

Tasks:

1. Show why this grammar is **not** LR(0).
2. Identify where shift/reduce or reduce/reduce conflicts arise.
3. Explain why the LR(0) parser cannot decide which production to apply.

8.11 Summary

Concept	Summary
LR(0) Item	Production with dot showing progress
Closure	Expand non-terminals after dot
GOTO	Move dot to the right
Canonical Items	All item sets form DFA states
ACTION/GOTO	Parsing table maps state + input to action
Conflicts	Shift/Reduce or Reduce/Reduce when table is ambiguous

8.12 C Programming Problems

Problem 1: Display LR(0) Items

C Code:

```
#include <stdio.h>
int main() {
    printf("Grammar: S->aSb|c\n");
    printf("LR(0) Items:\n");
    printf("S->•aSb | •c\n");
    printf("S->a•Sb\n");
    printf("S->aS•b\n");
    printf("S->aSb•\n");
    printf("S->c•\n");
    return 0;
}
```

Explanation:

- LR(0) items represent the grammar productions with a dot (·) indicating the position of the parser.
- Initially, the dot is before the first symbol in the production.
- As the parser reads symbols, the dot moves to the right.

- For `S -> aSb | c`, the items are:
  - $S \to \cdot aSb$
  - $S \to \cdot c$
  - $S \to a \cdot Sb$
  - $S \to aS \cdot b$
  - $S \to aSb\cdot$
  - $S \to c\cdot$

Problem 2: Show ACTION and GOTO Table

C Code:

```
#include <stdio.h>
int main() {
    printf("ACTION and GOTO Table for S->aSb|c:\n");
    printf("State |  a  |  b  |  c  |  $  |  S\n");
    printf("-----------------------------------\n");
    printf(" 0    | s1  |     | s2  |     | 3\n");
    printf(" 1    | s1  |     | s2  |     | 4\n");
    printf(" 2    |     |     |     | acc |\n");
    return 0;
}
```

Explanation:

- **ACTION** table defines shifts (s), reductions (r), accept (acc) or errors for terminals.
- **GOTO** table defines the transitions on non-terminals.
- For this grammar:
  - State 0: on 'a' shift to 1, on 'c' shift to 2, on 'S' goto 3.
  - State 1: on 'a' shift to 1, on 'c' shift to 2, on 'S' goto 4.
  - State 2: accept on end of input ($).

Problem 3: Parse Input String

C Code:

```
#include <stdio.h>
int main() {
    char str[10];
```

```
    printf("Enter input (like acb): ");
    scanf("%s", str);
    printf("Shift a\nShift c\nReduce S->c\nShift b\nReduce S->aSb\nAccept\n");
    return 0;
}
```

Explanation:

- Input string `acb` is parsed as:
  1. Shift `a`
  2. Shift `c`
  3. Reduce by $S \to c$
  4. Shift `b`
  5. Reduce by $S \to aSb$
  6. Accept the input string
- This simulates LR parsing steps based on the ACTION and GOTO tables.

Part II

Intermediate Representation & Optimization

Chapter 9

Intermediate Code Generation

9.1 Introduction to Intermediate Code Generation

The compiler is a bridge between a high-level programming language (like C or Java) and the machine's binary instructions. Direct translation into machine code is difficult due to:

- Architectural differences (x86, ARM, RISC-V).
- Complexity of high-level optimizations.
- Structural differences between source code and hardware execution.

To address this, compilers use an **Intermediate Representation (IR)**.

Key Goals of IR

1. **Machine Independence**: Same IR can map to different targets.
2. **Easy Optimization**: Enables common subexpression elimination, constant folding, dead code elimination, etc.
3. **Balance of Abstraction**: Close to machine code, but abstract enough for analysis.

Example:

```
x = (a + b) * (c - d);
```

Intermediate Representation (TAC):

```
t1 = a + b
t2 = c - d
x = t1 * t2
```

Hypothetical Assembly:

```
LDF R1, a
LDF R2, b
ADD R3, R1, R2
LDF R4, c
LDF R5, d
SUB R6, R4, R5
MUL R7, R3, R6
STF x, R7
```

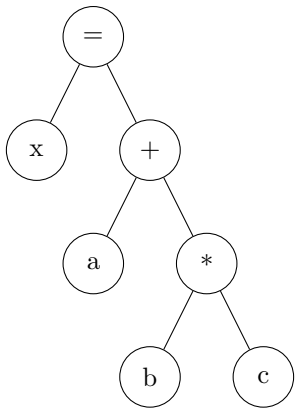
9.2 Desirable Properties of a Good Intermediate Representation

1. **Simplicity**: Easy generation from syntax trees.
2. **Accuracy**: Captures all details including precedence and types.
3. **Optimizable**: Supports efficient detection of redundancies.
4. **Portability**: Translatable to multiple architectures.
5. **Efficiency**: Should not be overly abstract.

9.3 Forms of Intermediate Representations

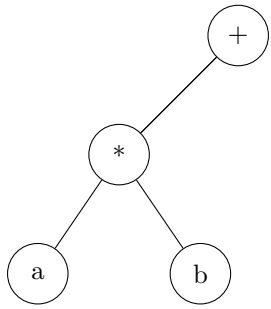
Syntax Trees

Expression: $x = a + b \times c$



Directed Acyclic Graphs (DAGs)

Expression: $x = a \times b + a \times b$



Three-Address Code (TAC)

For $(a + b) \times (c + d)$:

```
t1 = a + b
t2 = c + d
t3 = t1 * t2
```

9.4 Types of Three Address Code Instructions

```
// Assignment
x = y op z

// Unary Operation
x = op y
```

```
// Copy
x = y

// Jumps
goto L1
if x < y goto L2

// Procedure Calls
call p, n
return x
```

9.5 TAC Generation for Arithmetic Expressions

Example: (a+b)/(c+d-e)

```
t1 = a + b
t2 = c + d
t3 = t2 - e
t4 = t1 / t3
```

9.6 Boolean Expressions and Conditional TAC

Example:

```
if (a < b) then c = a else c = b
```

TAC:

```
if a < b goto L1
goto L2
L1: c = a
goto L3
L2: c = b
L3:
```

9.7 Flow Graphs and Basic Blocks

Example Program

```
t1 = a + b
if t1 < c goto L1
x = 5
goto L2
L1: x = 10
L2: return x
```

Basic Blocks:

1. B1: $t1 = a + b$
2. B2: if $t1 < c$ goto L1
3. B3: $x = 5$
4. B4: $x = 10$
5. B5: return x

9.8 DAG Construction in Depth

Expression: $x = (a \times b) + (a \times b) - c$

```
t1 = a * b
t2 = t1 + t1
t3 = t2 - c
x  = t3
```

9.9 Large Worked-Out Example

```
if (x < y) {
   z = x + y;
} else {
   z = x - y;
}
```

TAC:

```
if x < y goto L1
goto L2
L1: t1 = x + y
    z = t1
    goto L3
L2: t2 = x - y
    z = t2
L3:
```

9.10 Practice Exercises

1. Generate TAC for $(x + y) \times (z - w)/(p + q)$.
2. Convert into TAC: if $a \leq b$ then $x = y \times z$ else $x = y + z$.
3. Draw DAG and TAC for $m = (a * b) + (a * b) - c$.

4. Build a flow graph for: while $(i < 10)$ do if $j < k$ then $j = j + 1$ else $k = k - 1$.

9.11 Chapter Summary

- IR bridges high-level code and machine code.
- Syntax Trees capture structure; DAGs optimize repetitions.
- TAC is widely used, with simple instructions.
- Boolean and loop constructs handled via jumps.
- Flow Graphs show TAC control flow.

9.12 C Programming Problems

Problem 1: Generate TAC for Expression

```
#include <stdio.h>
int main() {
    printf("Expression: (a+b)*(c-d)\n");
    printf("TAC:\n");
    printf("t1 = a + b\n");
    printf("t2 = c - d\n");
    printf("t3 = t1 * t2\n");
    return 0;
}
```

Problem 2: TAC for If Statement

```
#include <stdio.h>
int main() {
    printf("if a < b then x = a else x = b\n");
    printf("TAC:\n");
    printf("if a < b goto L1\n");
    printf("x = b\n");
    printf("goto L2\n");
    printf("L1: x = a\n");
    printf("L2:\n");
    return 0;
}
```

Problem 3: TAC for While Loop

```
#include <stdio.h>
int main() {
    printf("while i < n do i = i + 1\n");
    printf("TAC:\n");
    printf("L1: if i < n goto L2\n");
    printf("goto L3\n");
    printf("L2: i = i + 1\n");
    printf("goto L1\n");
    printf("L3:\n");
    return 0;
}
```

Chapter 10

Control Flow Analysis

10.1 Introduction

Control Flow Analysis (CFA) is one of the most critical phases in the middle-end of a compiler. It provides the foundation for program optimizations, error detection, and code generation. CFA examines how control flows through a program — that is, the sequence in which instructions, conditions, and loops are executed.

Unlike lexical or syntax analysis, CFA looks at the entire structure of the program beyond tokens and grammar rules. Its primary data structure is the Control Flow Graph (CFG), a directed graph that models the paths a program may take during execution.

Why is CFA Important?

- Detect unreachable code: Helps find code that can never execute.
- Build flow graphs: Required for optimization like constant propagation.
- Loop handling: Detects natural loops for loop optimizations.
- Dead variable analysis: Helps in register allocation.
- Error detection: Can detect infinite loops or missing return statements.

10.2 Basic Blocks

Definition: A basic block is a straight-line sequence of code where:

- Control enters only at the beginning
- Control leaves only at the end
- No jumps or labels appear in the middle

Features:

- **Single Entry & Exit**: ensures predictability
- **Atomic Execution**: either all statements execute or none
- **Simplification**: analyzing the flow between blocks is easier than analyzing each line

Example Program

```
a = b + c;
d = a * e;
if (d < f) goto L1;
g = d - f;
L1: h = g + 1;
```

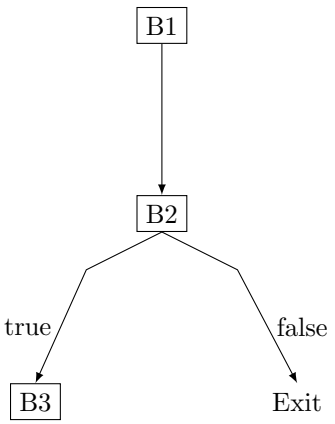
Step 1: Identify Leaders

- Line 1 (first statement)
- Line 3 (target of conditional jump)
- Line 5 (label L1)

Step 2: Form Basic Blocks

- B1: a = b + c; d = a * e;
- `B2: if (d < f) goto L1; g = d - f;`
- B3: h = g + 1;

Step 3: Draw CFG



Practice Exercise: Break the following into basic blocks:

```
1: x = 5
2: if (y > 0) goto L1
3: y = y - 1
4: goto L2
5: L1: y = y + 2
6: L2: z = y * x
```

10.3 Leaders: The Entry Points

Rules Recap:

- First statement is a leader.
- The target of any jump is a leader.
- The instruction following a conditional or unconditional jump is a leader.

Example:

```
1: t1 = a + b
2: if t1 < c goto 5
3: d = t1 - e
4: goto 6
5: d = e * f
6: return d
```

Leaders: 1, 5, 6

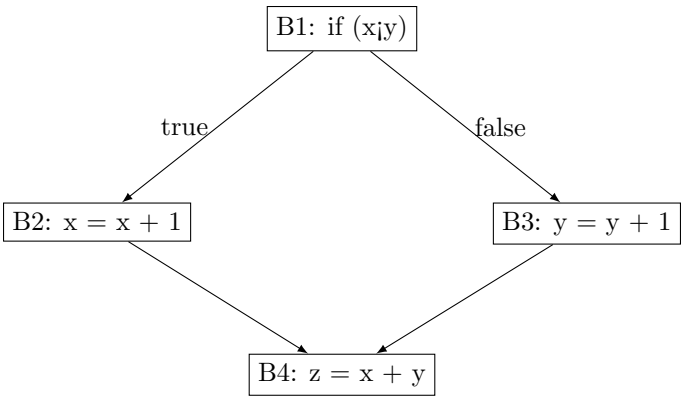
10.4 Control Flow Graph (CFG) in Detail

A Control Flow Graph (CFG) is a directed graph where:

- Nodes = basic blocks
- Edges = flow of control between blocks

Example: If-Else

```
if (x < y)
   x = x + 1;
else
   y = y + 1;
z = x + y;
```



10.5 Properties of Control Flow Graph

A CFG has some important properties that are useful for compiler optimizations:

- **Entry and Exit Nodes:** CFG has a unique entry node where control starts and a unique exit node where control finishes.
- **Reachability:** Every node should be reachable from the entry node; unreachable nodes indicate dead code.
- **Paths:** A path in a CFG is a sequence of nodes connected by edges. Paths represent possible execution flows.
- **Loops:** Loops form cycles in the CFG, which need special handling during optimization.

10.6 Loop Detection and Natural Loops

Loops are a fundamental control structure, and detecting them is crucial for optimizations like loop-invariant code motion, induction variable elimination, and strength reduction.

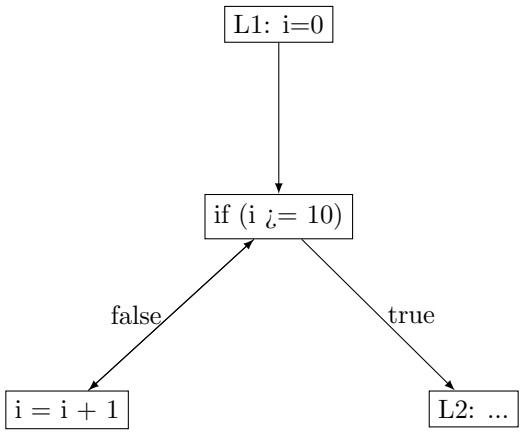
Natural Loop Definition: A natural loop is a set of nodes that have a single entry point called the loop header and contain at least one back edge.

Back Edge: An edge from a node to one of its ancestors in the CFG (i.e., an edge that points backward in the flow).

Example:

```
L1: i = 0;
    if (i >= 10) goto L2;
    i = i + 1;
    goto L1;
L2: ...
```

CFG Representation:



The edge from ‘Body‘ back to ‘Cond‘ is a back edge, indicating a loop.

10.7 Data Flow Analysis: Overview

Control Flow Analysis often serves as a basis for *Data Flow Analysis* (DFA), which focuses on how data values propagate through the CFG. DFA enables many optimizations and error detections such as:

- Constant propagation
- Live variable analysis
- Available expressions
- Reaching definitions

These analyses are iterative and use the CFG to calculate data properties at each program point.

Next Steps: In the upcoming chapters, we will explore data flow equations, worklist algorithms, and classic data flow problems in detail.

10.8 Dominators

Definition: A node d dominates node n if every path from the entry node to n must go through d.

Properties:

- Every node dominates itself.
- The entry node dominates all nodes.

- Dominator trees represent the dominance relationship compactly.

Example:

```
Entry -> A -> B -> C
           \-> D -> C
```

Dominators of C: Entry, A, and C.

10.9 Post-Dominators

Definition: Node p post-dominates node n if every path from n to the exit node must go through p.

Usage: Post-dominators help in understanding control dependencies and are used in code optimization and exception handling.

10.10 Control Dependence

A statement S is control dependent on a branch B if the outcome of B decides whether S executes.

Control Dependence Graph (CDG): This graph shows dependencies between branches and statements.

10.11 Exception Handling in CFA

Exception handling introduces non-linear control flows due to abrupt jumps (exceptions).

- CFG must include edges to exception handlers.
- Exception edges complicate analysis but are critical for correctness.

10.12 Summary and Exercises

Summary:

- CFA uses CFG to represent all possible execution paths.
- Basic blocks simplify CFG nodes.
- Leaders define block entry points.
- Dominators and post-dominators explain hierarchical control flow.
- Control dependence determines which statements depend on branches.

- Exception handling affects control flow and must be incorporated.

Practice Exercise:

```
Analyze the following program:

1: start:
2: x = 0
3: if (y > 0) goto L1
4: x = x + 1
5: goto L2
6: L1: x = x - 1
7: L2: return x

Tasks:
(a) Identify basic blocks and leaders.
(b) Draw the CFG including exception edges if any.
(c) Determine dominators and post-dominators.
(d) Identify control dependencies.
```

10.13 C Programming Problems

Problem 1: Build Basic Blocks

```
#include <stdio.h>
int main() {
    printf("Code: t1=a+b; if(t1<c) goto L1; x=5; L1: y=10;\n");
    printf("Basic Blocks:\n");
    printf("B1: t1=a+b\n");
    printf("B2: if(t1<c) goto L1\n");
    printf("B3: x=5\n");
    printf("B4: y=10\n");
    return 0;
}
```

Problem 2: Draw CFG

```
#include <stdio.h>
int main() {
    printf("CFG:\n");
    printf("B1 -> B2\n");
    printf("B2 -> B3 (false)\n");
    printf("B2 -> B4 (true)\n");
    return 0;
}
```

Problem 3: Detect Loop Back Edge

```
#include <stdio.h>
int main() {
    printf("Loop Example: while(i<n) i=i+1;\n");
    printf("Back Edge: from update block back to condition\n");
    return 0;
}
```

Chapter 11

Code Optimization

11.1 Introduction

The purpose of a compiler is not only to translate a high-level program into machine code but also to ensure that the generated code is efficient. The quality of the generated code is a significant factor in the program's performance. This is where code optimization comes into play.

What is Code Optimization?

Code Optimization is the process of improving the intermediate or final code produced by a compiler so that it runs faster, uses less memory, or consumes fewer resources—without changing the program's behavior.

Key Points:

- Optimization does not change the program's semantics. The output of the program must remain the same.
- Optimizations may target speed, space, power consumption, or a combination.

Why Optimization Matters

```
t1 = a * b
t2 = a * b
x = t1 + t2
```

Optimized version:

```
t1 = a * b
x = t1 + t1
```

This saves one multiplication instruction. Even small savings inside loops can drastically reduce execution time.

11.2 Objectives of Code Optimization

The objectives go beyond just making the code faster:

1. Execution Speed

- Minimize CPU cycles.
- Reduce costly operations like multiplications/divisions.
- Rearrange instructions to avoid pipeline stalls.

2. Memory Efficiency

- Remove unused variables.
- Keep variables in registers instead of memory.
- Eliminate redundant loads and stores.

3. Energy and Power Saving

Optimizations aim to reduce power consumption—important for mobile and embedded systems.

4. Maintainability of Generated Code

Optimized code should remain readable for debugging if intermediate outputs are generated.

5. Balance Between Compile Time and Execution Time

- Light optimizations → fast compile, reasonable run time.
- Aggressive optimizations (e.g., -O3 in GCC) → longer compile, faster execution.

11.3 Classification of Optimizations

Optimizations can be classified along several dimensions.

11.3.1 Machine-Independent Optimizations

These do not depend on the final target machine. They operate on Intermediate Representation (IR).

- Constant Folding
- Common Subexpression Elimination
- Dead Code Elimination
- Loop Optimizations

11.3.2 Machine-Dependent Optimizations

These depend on the instruction set and architecture.

- Register Allocation
- Instruction Scheduling
- Peephole Optimization

11.3.3 Local vs. Global Optimizations

- Local Optimization: Within a single basic block.
- Global Optimization: Across multiple basic blocks using the Control Flow Graph (CFG).

Practice Exercise:

```
x = 10
y = x + 0
if (y < 20) goto L1
```

Classify the possible optimizations. Are they machine-independent or dependent?

11.4 Common Machine-Independent Optimizations

11.4.1 Constant Folding

Replace expressions with computed values.

```
x = 4 * 2    -->  x = 8
t = (3 * 5) + (6 / 2)   -->  t = 18
```

11.4.2 Constant Propagation

```
x = 5
y = x + 2   -->   y = 7
```

11.4.3 Dead Code Elimination

```
a = 10
b = 20
a = 15
print(b)
```

Optimized:

```
b = 20
a = 15
print(b)
```

11.4.4 Common Subexpression Elimination (CSE)

```
t1 = a + b
t2 = a + b   -->  t2 = t1
```

Extended:

```
t1 = a * b
t2 = c + d
t3 = a * b + t2
t4 = a * b - t2
```

Optimized:

```
t1 = a * b
t2 = c + d
t3 = t1 + t2
t4 = t1 - t2
```

11.4.5 Strength Reduction

```
x = y * 8   -->   x = y << 3
```

11.4.6 Loop Invariant Code Motion

```
for i=1 to n:
    x = a + b
    y = x + i
```

Optimized:

```
x = a + b
for i=1 to n:
    y = x + i
```

11.4.7 Loop Unrolling

```
for i=0; i<4; i++:
   sum = sum + arr[i]
```

Optimized:

```
sum = sum + arr[0]
sum = sum + arr[1]
sum = sum + arr[2]
sum = sum + arr[3]
```

Practice Task: Apply optimizations to:

```
for i=0; i<10; i++:
    x = 2 * 3
    y = a + b
    z = x + y
```

11.5 Peephole Optimization (In-Depth)

Peephole optimization uses a small sliding window of instructions to remove local inefficiencies.

Techniques

- Redundant Instruction Removal
- Algebraic Simplification
- Strength Reduction
- Unreachable Code Elimination

Example: Before:

```
LDF R1, a
MULF R1, R1, 1
STF a, R1
```

After:

```
LDF R1, a
STF a, R1
```

11.6 Large Worked Example

Given:

```
for (i=0; i<100; i++) {
   t1 = a * b;
   x = t1 + i;
   y = a * b;
}
```

Step 1: Detect CSE
Step 2: Apply Loop Invariant Motion

```
t1 = a * b
for (i=0; i<100; i++) {
   x = t1 + i
   y = t1
}
```

11.7 Exercises

1. Optimize:

   ```
   for (i=0; i<10; i++) {
      x = y + z;
      w = x + i;
   }
   ```

2. Remove dead code:

   ```
   a = 5
   b = 10
   a = 7
   print(b)
   ```

3. Peephole Optimize:

   ```
   MOV R1, R1
   ADD R2, R3, 0
   MUL R4, R5, 1
   ```

4. Loop Optimize:

   ```
   while (i < 100) {
      z = 5 * 2
      i = i + 1
   }
   ```

11.8 Summary

- Code Optimization improves speed, memory usage, and efficiency.
- Machine-independent techniques include constant folding, CSE, and loop optimizations.
- Machine-dependent techniques include register allocation and peephole optimization.
- Loops are the primary focus of optimization.
- Peephole optimization provides small but critical local improvements.

11.9 C Programming Problems

Problem 1: Constant Folding

```
#include <stdio.h>
int main() {
    printf("Code: x = 4+6\n");
    printf("Optimized: x = 10\n");
    return 0;
}
```

Problem 2: Dead Code Elimination

```
#include <stdio.h>
int main() {
    printf("Code:\n x=5; y=10; x=7; print(y);\n");
    printf("Optimized:\n y=10; x=7; print(y);\n");
    return 0;
}
```

Problem 3: Loop Invariant Code Motion

```
#include <stdio.h>
int main() {
    printf("Before:\n for i=1 to n: x=a+b; y=x*i;\n");
    printf("After:\n x=a+b; for i=1 to n: y=x*i;\n");
    return 0;
}
```

Part III

Compiler Lab Project Toolkit

Chapter 12

Project: *Compiler Toolkit*

12.1 Project Overview

The **Web-Based Compiler Toolkit** is a lightweight yet powerful browser-based application that brings core compiler design concepts into an interactive, real-time environment. Instead of writing code and manually calculating results on paper, students can use this toolkit to instantly test their grammars, analyze language properties, and even generate intermediate code.

The toolkit acts as a bridge between compiler theory and practical application. Its purpose is not only to save time but also to improve conceptual understanding by providing immediate feedback and results for:

- Grammar analysis (`FIRST` and `FOLLOW` sets)
- Left recursion elimination
- Left factoring transformation
- Regular expression pattern matching
- Three-Address Code (TAC) generation

This tool supports both beginners and advanced learners:

- **Beginners** can learn by using preloaded default inputs that demonstrate each concept.
- **Advanced users** can test custom grammars and expressions for deeper experimentation.

12.2 Motivation

When learning Compiler Design, students often face a common problem:

> "The theory makes sense in class, but when I try to apply it to a grammar or problem, I get stuck."

Manual computation of `FIRST/FOLLOW` sets, left recursion removal, or intermediate code generation can be time-consuming and error-prone, especially during exams or lab work.

Our motivation for building this toolkit was:

- Bringing automation to repetitive compiler theory calculations.
- Providing visual and interactive learning instead of static textbook examples.
- Allowing custom experimentation without requiring local compiler installations.
- Creating a tool that works on any device — laptops, desktops, or even mobile phones.

12.3 Features

The Web-Based Compiler Toolkit currently offers six major features:

12.3.1 FIRST and FOLLOW Computation

- Computes the `FIRST` set for all non-terminals in a grammar.
- Computes the `FOLLOW` set for all non-terminals.
- Works for any user-provided grammar in the correct format.
- Includes a default example grammar for quick testing.

Example:

```
Input Grammar:
S -> A B
A -> a A |
B -> b B |

Output:
FIRST(S) = { a, b,  }
FOLLOW(S) = { $ }
```

12.3.2 Left Recursion Detection & Elimination

- Detects immediate and indirect left recursion.
- Automatically transforms the grammar into left recursion–free form.

Example:

```
Input: E -> E + T | T
Output:
E -> T E'
E' -> + T E' |
```

12.3.3 Left Factoring

- Identifies whether common prefixes exist in productions.
- Generates a factored version of the grammar for LL(1) parsing compatibility.

Example:

```
Input: S -> abcA | abcB
Output:
S -> abcS'
S' -> A | B
```

12.3.4 Regular Expression Matching

- Accepts custom regular expressions from the user.
- Allows the user to enter sample text for testing.
- Highlights all matching patterns in the output.
- Preloaded with a default regex example.

12.3.5 Three-Address Code (TAC) Generation

- Accepts grammar input and generates intermediate representation in TAC format.
- Shows step-by-step breakdown of the translation.

Example:

```
Input Grammar: E -> E + T | T
Input Expression: a + b * c
Output TAC:
t1 = b * c
t2 = a + t1
```

12.3.6 Custom Grammar & Input Handling

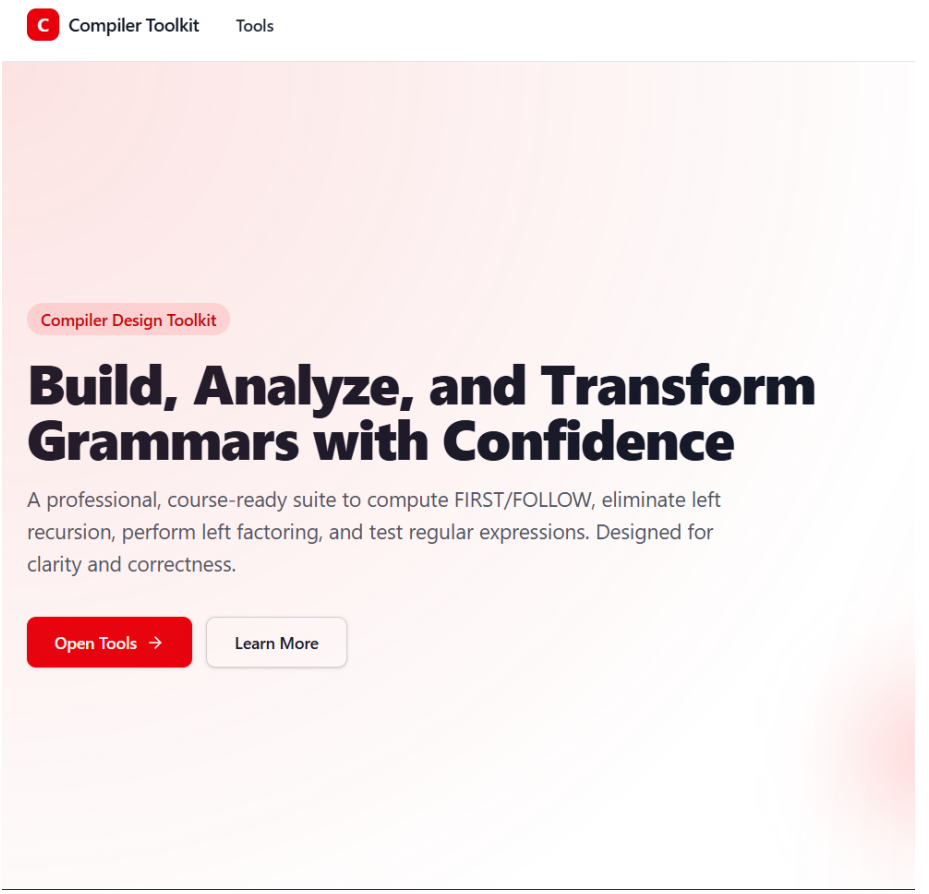
-The toolkit does not force fixed inputs , users can modify and test any grammar they want. -Each section has default placeholders to guide new users.

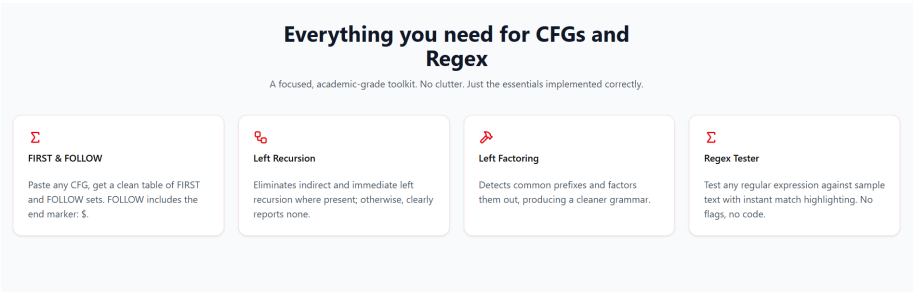
12.4 Interface Snapshots

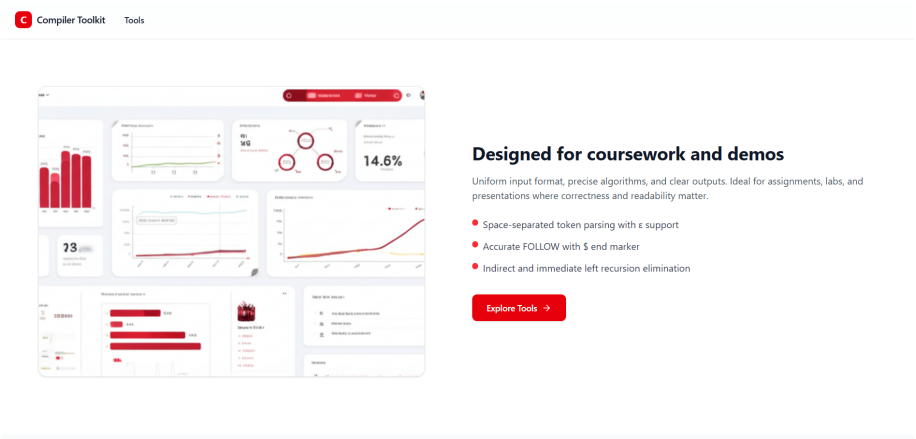
The interface is built with **Tailwind CSS** for a clean, responsive design. Each feature is presented in its own section or tab, with:

- A grammar input box (multiline, scrollable).
- Buttons to compute, analyze, or transform grammar.
- A results panel showing structured outputs.

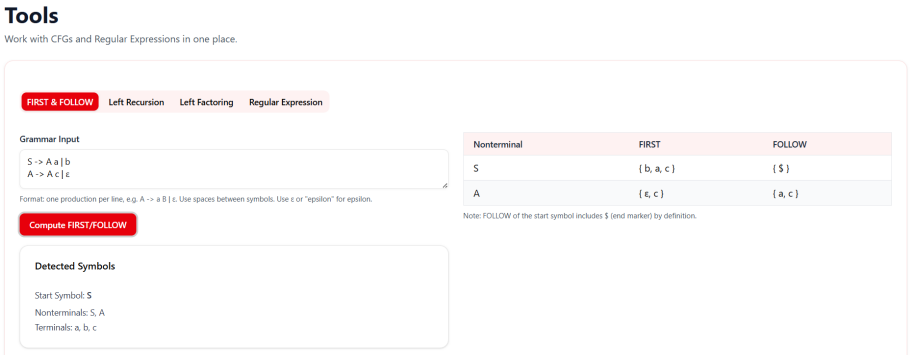
Screenshots :


C Compiler Toolkit
Tools
Compiler Design Toolkit
Build, Analyze, and Transform Grammars with Confidence
A professional, course-ready suite to compute FIRST/FOLLOW, eliminate left recursion, perform left factoring, and test regular expressions. Designed for clarity and correctness.
Open Tools →
Learn More

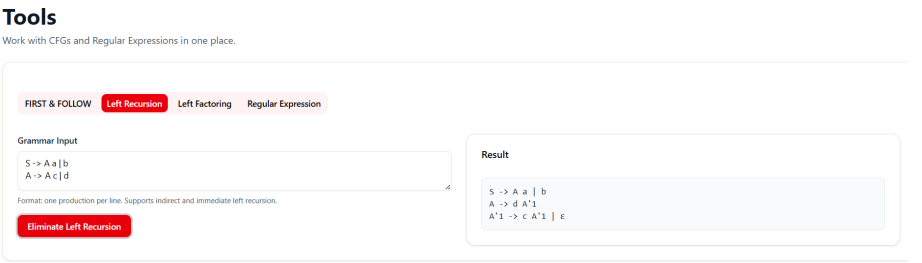

Everything you need for CFGs and Regex
A focused, academic-grade toolkit. No clutter. Just the essentials implemented correctly.
FIRST & FOLLOW
Paste any CFG, get a clean table of FIRST and FOLLOW sets. FOLLOW includes the end marker: $.
Left Recursion
Eliminates indirect and immediate left recursion where present; otherwise, clearly reports none.
Left Factoring
Detects common prefixes and factors them out, producing a cleaner grammar.
Regex Tester
Test any regular expression against sample text with instant match highlighting. No flags, no code.


Compiler Toolkit
Tools
14.6%
Designed for coursework and demos
Uniform input format, precise algorithms, and clear outputs. Ideal for assignments, labs, and presentations where correctness and readability matter.
Space-separated token parsing with ε support
Accurate FOLLOW with $ end marker
Indirect and immediate left recursion elimination
Explore Tools

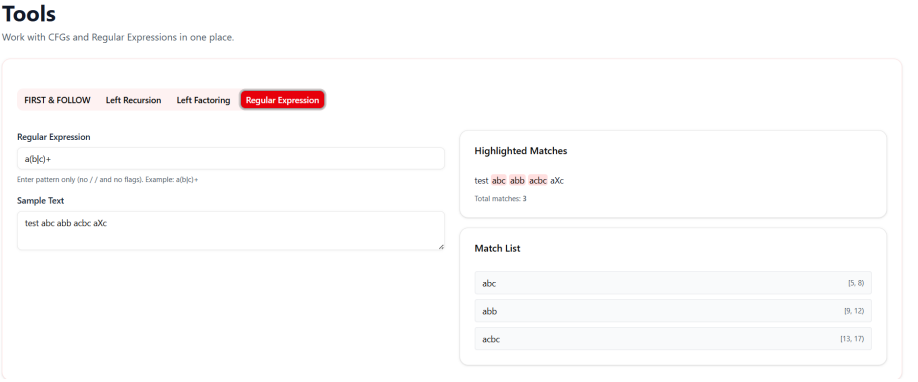
- FIRST/FOLLOW computation results.


Tools
Work with CFGs and Regular Expressions in one place.
FIRST & FOLLOW
Left Recursion
Left Factoring
Regular Expression
Grammar Input
S -> A a | b
A -> A c | ε
Format: one production per line, e.g. A -> a B | ε. Use spaces between symbols. Use ε or "epsilon" for epsilon.
Compute FIRST/FOLLOW
Detected Symbols
Start Symbol: S
Nonterminals: S, A
Terminals: a, b, c
Nonterminal
FIRST
FOLLOW
S
{ b, a, c }
{ $ }
A
{ ε, c }
{ a, c }
Note: FOLLOW of the start symbol includes $ (end marker) by definition.

- Left recursion elimination results.


Tools
Work with CFGs and Regular Expressions in one place.
FIRST & FOLLOW
Left Recursion
Left Factoring
Regular Expression
Grammar Input
S -> A a | b
A -> A c | d
Format: one production per line. Supports indirect and immediate left recursion.
Eliminate Left Recursion
Result
S -> A a | b
A -> d A'1
A'1 -> c A'1 | ε

- Regular expression match highlights.



- TAC generation output.

12.5 Tools/Technologies Used

- **HTML5** — Base structure for the web app.
- **Tailwind CSS** — Styling and responsive design.
- **Next.js** — Frontend framework for efficient routing and rendering.
- **JavaScript (ES6+)** — Logic for parsing grammars, computing sets, and generating TAC.
- **GitHub** — Version control and collaboration.
- **Online Deployment** — Accessible from any browser without installation.

12.6 GitHub Repository / Deployment Link

Repository:

Figure 12.1: Github Compiler Tool kit QR code

A QR code embedded here that links directly to the live project for hands-on use:

Figure 12.2: Compiler Tool kit QR code

12.7 Sample Use Cases

Case 1: Beginner testing FIRST/FOLLOW: Loads default grammar → Clicks `Compute` → Instantly sees results.
Case 2: Advanced grammar testing: Modifies grammar to a personal assignment → Checks for recursion and factoring.
Case 3: TAC generation: Inputs a simple expression grammar → Gets step-by-step TAC output.

12.8 Flowchart / Diagram of Process

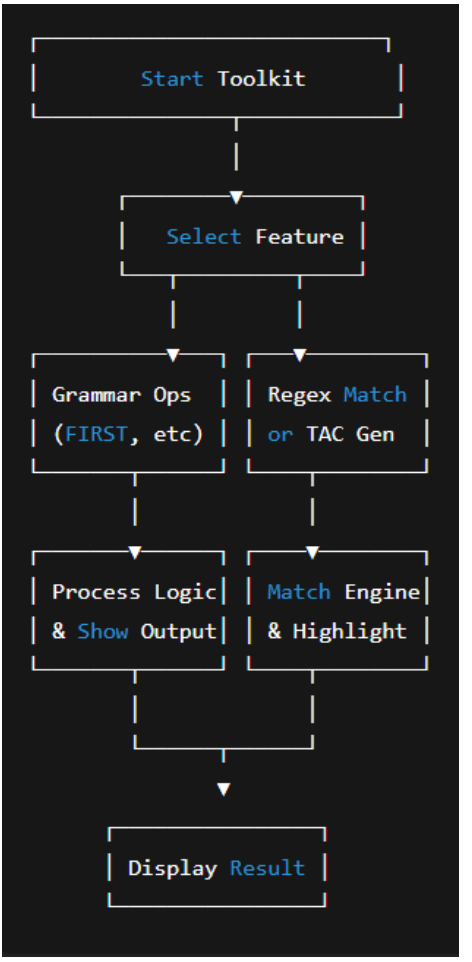


Figure 12.3: Flowchart

Part IV

Lab Work Compilation

Chapter 13

Lab Sheet Solutions (With C Code)

13.1 Lab Sheet 1: Program to Count the Length of a String

```
#include <stdio.h>
int main() {
    char str[100];
    int length = 0;
    printf("Enter a string: ");
    scanf("%[^\n]", str);

    while (str[length] != '\0') {
        length++;
    }
    printf("Length of the string: %d\n", length);
    return 0;
}
```

Input: Hello World
Output: Length of the string: 11

13.2 Lab Sheet 2: Count the Number of White Spaces in a String

```
#include <stdio.h>
int main() {
    char str[200];
```

```
    int i = 0, count = 0;
    printf("Enter a string: ");
    scanf("%[^\n]", str);
    while (str[i] != '\0') {
        if (str[i] == ' ')
            count++;
        i++;
    }
    printf("Number of white spaces: %d\n", count);
    return 0;
}
```

Input: This is a test
Output: Number of white spaces: 3

13.3 Lab Sheet 3: Remove White Spaces from a String

```
#include <stdio.h>
int main() {
    char str[200], result[200];
    int i = 0, j = 0;
    printf("Enter a string: ");
    scanf("%[^\n]", str);
    while (str[i] != '\0') {
        if (str[i] != ' ')
            result[j++] = str[i];
        i++;
    }
    result[j] = '\0';
    printf("String without spaces: %s\n", result);
    return 0;
}
```

Input: Hello World Program
Output: String without spaces: HelloWorldProgram

13.4 Lab Sheet 4: Multiple Line Input & Output

```
#include <stdio.h>
int main() {
    char str[500];
```

```
    printf("Enter multiple lines (Press Ctrl+D or Ctrl+Z to stop):\n");
    while (scanf("%[^\n]%*c", str) != EOF) {
        printf("You entered: %s\n", str);
    }
    return 0;
}
```

Input: Compiler Design
Lab Work
Output: You entered: Compiler Design
You entered: Lab Work

13.5 Lab Sheet 5: Extract Single Line Comments from String

```
#include <stdio.h>
int main() {
    char str[200];
    int i = 0;
    printf("Enter a line with comments: ");
    scanf("%[^\n]", str);
    while (str[i] != '\0') {
        if (str[i] == '/' && str[i+1] == '/') {
            printf("Single line comment: %s\n", &str[i]);
            break;
        }
        i++;
    }
    return 0;
}
```

Input: int a = 10; // This is a comment
Output: Single line comment: // This is a comment

13.6 Lab Sheet 6: Count Number of Lines

```
#include <stdio.h>
int main() {
    char str[500];
    int i = 0, count = 1;
    printf("Enter text (end with ENTER twice):\n");
    while (scanf("%[^\n]%*c", str) != EOF) {
        count++;
    }
```

```
    printf("Number of lines: %d\n", count - 1);
    return 0;
}
```

Input:
Hello
World
Output:
Number of lines: 2

13.7 Lab Sheet 7: Tokenize Input String (Without strtok)

```
#include <stdio.h>
int main() {
    char ch, word[100];
    int id = 0;
    printf("Enter a sentence: ");
    while (1) {
        ch = getchar();
        if (ch == '\n') break;
        if ((ch >= 'A' && ch <= 'Z') || (ch >= 'a' && ch <= 'z')) {
            word[id++] = ch;
        } else {
            if (id > 0) {
                word[id] = '\0';
                printf("%s\n", word);
                id = 0;
            }
            if (ch != ' ') printf("%c\n", ch);
        }
    }
    if (id > 0) {
        word[id] = '\0';
        printf("%s\n", word);
    }
    return 0;
}
```

Input: Hello, World!
Output:
Hello
,
World
!

13.8 Lab Sheet 8: Identify and Count Articles

```
#include <stdio.h>
#define MAX 1000
char toLower(char ch) {
    if (ch >= 'A' && ch <= 'Z')
        return ch + 32;
    return ch;
}
int compareIgnoreCase(char *s1, char *s2) {
    int i = 0;
    while (s1[i] != '\0' && s2[i] != '\0') {
        if (toLower(s1[i]) != toLower(s2[i]))
            return 0;
        i++;
    }
    return (s1[i] == '\0' && s2[i] == '\0');
}
int isDelimiter(char ch) {
    return (ch == ' ' || ch == '.' || ch == ',' || ch == '?'
    || ch == '!' || ch == ';' || ch == ':' || ch == '\n' || ch == '\t');
}
int main() {
    char input[MAX];
    char word[50];
    int aCount = 0, anCount = 0, theCount = 0;
    printf("Enter a sentence:\n");
    scanf(" %[^\n]", input);
    int i = 0, j = 0;
    while (input[i] != '\0') {
        if (!isDelimiter(input[i])) {
            word[j++] = input[i];
        } else {
            if (j > 0) {
                word[j] = '\0';
                if (compareIgnoreCase(word, "a")) aCount++;
                else if (compareIgnoreCase(word, "an")) anCount++;
                else if (compareIgnoreCase(word, "the")) theCount++;
                j = 0;
            }
        }
        i++;
    }
    if (j > 0) {
        word[j] = '\0';
```

```
        if (compareIgnoreCase(word, "a")) aCount++;
        else if (compareIgnoreCase(word, "an")) anCount++;
        else if (compareIgnoreCase(word, "the")) theCount++;
    }
    printf("\nArticle Counts:\n");
    printf("A - %d\n", aCount);
    printf("An - %d\n", anCount);
    printf("The - %d\n", theCount);
    return 0;
}
```

Input:
An ant bit the boy and a bird saw the event.
Output:
Article Counts:
A - 1
An - 1
The - 2

13.9 Lab Sheet 9: Identifier Validity and Keyword Check

```
#include <stdio.h>
char keywords[32][10] = {
    "auto", "break", "case", "char", "const", "continue", "default", "do",
    "double", "else", "enum", "extern", "float", "for", "goto", "if",
    "int", "long", "register", "return", "short", "signed", "sizeof", "static",
    "struct", "switch", "typedef", "union", "unsigned", "void", "volatile",
    "while"
};
int stringComp(char a[], char b[]) {
    int i = 0;
    while (a[i] != '\0' && b[i] != '\0') {
        if (a[i] != b[i]) return 0;
        i++;
    }
    return (a[i] == '\0' && b[i] == '\0');
}
int isKeyword(char str[]) {
    int i;
    for (i = 0; i < 32; i++) {
        if (stringComp(str, keywords[i])) {
            return 1;
        }
```

```
    }
    return 0;
}
int isAlphabet(char ch) {
    return ((ch >= 'A' && ch <= 'Z') || (ch >= 'a' && ch <= 'z'));
}
int isDigit(char ch) {
    return (ch >= '0' && ch <= '9');
}
int isValidId(char str[]) {
    int i;
    if (!(isAlphabet(str[0]) || str[0] == '_')) return 0;
    for (i = 1; str[i] != '\0'; i++) {
        if (!(isAlphabet(str[i]) || isDigit(str[i]) || str[i] == '_'))
        return 0;
    }
    if (isKeyword(str)) return 0;
    return 1;
}
int main() {
    char input[100];
    printf("Enter a string: ");
    scanf("%s", input);

    if (isKeyword(input)) {
        printf("'%s' is a keyword.\n", input);
    } else if (isValidId(input)) {
        printf("'%s' is a valid identifier.\n", input);
    } else {
        printf("'%s' is an invalid identifier.\n", input);
    }
    return 0;
}
```

```
Input: int
Output: 'int' is a keyword.

Input: my_var1
Output: 'my_var1' is a valid identifier.

Input: 1var
Output: '1var' is an invalid identifier.
```

13.10 Lab Sheet 10: FIRST() of Two Productions (C++)

```
#include <iostream>
#include <vector>
#include <map>
#include <set>
#include <sstream>
#include <algorithm>
using namespace std;
vector<string> splitChunks(const string& rhs) {
    vector<string> chunks;
    string currentChunk;
    for (size_t i = 0; i < rhs.size(); ++i) {
        if (rhs[i] == '|') {
            if (!currentChunk.empty()) {
                chunks.push_back(currentChunk);
                currentChunk.clear();
            }
        } else if (rhs[i] == ' ') {
            continue;
        } else {
            if (rhs.substr(i, 2) == "id") {
                currentChunk += "id";
                ++i;
            } else {
                currentChunk += rhs[i];
            }
        }
    }
    if (!currentChunk.empty()) {
        chunks.push_back(currentChunk);
    }
    return chunks;
}
bool isNonTerminal(const string& symbol, const map<string, vector<string>>&
grammar) { return grammar.find(symbol) != grammar.end();
}
void splitSymbols(const string& prod, vector<string>& symbols) {
    for (size_t i = 0; i < prod.size();) {
        if (prod.substr(i, 2) == "id") {
            symbols.push_back("id");
            i += 2;
        } else {
```

```
            symbols.push_back(string(1, prod[i]));
            i++;
        }
    }
}
void computeFirst(
    const string& symbol,
    const map<string, vector<string>>& grammar,
    map<string, set<string>>& firstSets,
    set<string>& visited
) {
    if (!isNonTerminal(symbol, grammar)) {
        firstSets[symbol].insert(symbol);
        return;
    }
    if (visited.count(symbol)) return;
    visited.insert(symbol);
    for (const string& prod : grammar.at(symbol)) {
        if (prod == "~") {
            firstSets[symbol].insert("~");
        } else {
            vector<string> symbols;
            splitSymbols(prod, symbols);
            bool epsilonChain = true;
            for (const string& sym : symbols) {
                computeFirst(sym, grammar, firstSets, visited);
                for (const string& f : firstSets[sym]) {
                    if (f != "~") firstSets[symbol].insert(f);
                }
                if (firstSets[sym].find("~") == firstSets[sym].end()) {
                    epsilonChain = false;
                    break;
                }
            }
            if (epsilonChain) {
                firstSets[symbol].insert("~");
            }
        }
    }
}
int main() {
    map<string, vector<string>> grammar;
    string line;
    cout << "Enter grammar rules (Empty line to stop):" << endl;
    while (getline(cin, line)) {
```

```
        if (line.empty()) break;

        size_t arrowPos = line.find("->");
        if (arrowPos == string::npos) {
            cout << "Invalid grammar format." << endl;
            continue;
        }
        string lhs = line.substr(0, arrowPos);
        lhs.erase(remove(lhs.begin(), lhs.end(), ' '), lhs.end());

        string rhs = line.substr(arrowPos + 2);
        vector<string> chunks = splitChunks(rhs);
        grammar[lhs] = chunks;
    }
    map<string, set<string>> firstSets;
    for (const auto& rule : grammar) {
        set<string> visited;
        computeFirst(rule.first, grammar, firstSets, visited);
    }
    for (const auto& rule : grammar) {
        cout << "First(" << rule.first << ") = {";
        size_t cnt = 0;
        for (const string& f : firstSets[rule.first]) {
            cout << f;
            if (++cnt < firstSets[rule.first].size()) cout << ", ";
        }
        cout << "}" << endl;
    }

    return 0;
}
```

```
Input:
E -> T E'
E' -> + T E' | ~
T -> F T'
T' -> * F T' | ~
F -> ( E ) | id

Output:
First(E) = ((, id)
First(E') = (+, ~)
First(T) = ((, id)
First(T') = (*, ~)
First(F) = ((, id)
```

13.11 Lab Sheet 11: Preposition Count

```
#include <stdio.h>
int comp(char a[], char b[]) {
    int i = 0;
    while (a[i] != '\0' && b[i] != '\0') {
        if (a[i] != b[i]) return 0;
        i++;
    }
    return (a[i] == '\0' && b[i] == '\0');
}
int main() {
    char str[300], words[100][20];
    char preps[][4][20] = {
        {"in"}, {"on"}, {"at"}, {"by"}, {"for"}, {"from"}, {"with"},
        {"about"},{"under"}, {"before"}, {"after"}, {"between"},
        {"in", "front", "of"}, {"in", "spite", "of"}, {"due", "to"},
        {"out", "of"},{"according", "to"}, {"because", "of"},
        {"instead", "of"}, {"prior", "to"},
        {"next", "to"}, {"on", "top", "of"}
    };
    int prepCount = sizeof(preps) / sizeof(preps[0]);
    printf("Enter a sentence:\n");
    scanf("%[^\n]s", str);
    int i = 0, j = 0, k = 0;
    while (str[i] != '\0') {
        if (str[i] != ' ') {
            words[k][j++] = str[i];
        } else {
            if (j > 0) {
                words[k][j] = '\0';
                k++;
                j = 0;
            }
        }
        i++;
    }
    if (j > 0) {
        words[k][j] = '\0';
        k++;
    }
    int count = 0;
    for (i = 0; i < k; i++) {
        for (int p = 0; p < prepCount; p++) {
            int match = 1;
```

```
            for (int w = 0; w < 3; w++) {
                if (preps[p][w][0] == '\0') break;
                if (!comp(words[i + w], preps[p][w])) {
                    match = 0;
                    break;
                }
            }
            if (match) {
                int skip = 0;
                while (preps[p][skip][0] != '\0') skip++;
                i += skip - 1;
                count++;
                break;
            }
        }
    }
    printf("Total prepositions found: %d\n", count);
    return 0;
}
```

Input: The book is on the table in front of the chair.
Output: Total prepositions found: 2

13.12 Lab Sheet 12: Max Frequency Word

```
#include <stdio.h>
int compare(char s1[], char s2[]) {
    int i = 0;
    while (s1[i] != '\0' && s2[i] != '\0') {
        if (s1[i] != s2[i]) return 0;
        i++;
    }
    return (s1[i] == '\0' && s2[i] == '\0');
}
int main() {
    char ch, words[50][20], temp[20];
    int i = 0, j = 0, k = 0, m, count[50] = {0};
    printf("Enter a string :\n");
    while (1) {
        ch = getchar();
        if (ch == ' ' || ch == '\n' || ch == '\t') {
            if (j > 0) {
                temp[j] = '\0';
                int n = 0;
                while (temp[n] != '\0') {
```

```
                    words[k][n] = temp[n];
                    n++;
                }
                words[k][n] = '\0';
                k++;
                j = 0;
            }
            if (ch == '\n') break;
        } else {
            temp[j++] = ch;
        }
    }
    for (i = 0; i < k; i++) {
        count[i] = 1;
        for (j = i + 1; j < k; j++) {
            if (compare(words[i], words[j])) {
                count[i]++;
                for (m = j; m < k - 1; m++) {
                    int n = 0;
                    while ((words[m][n] = words[m + 1][n]) != '\0') n++;
                    count[m] = count[m + 1];
                }
                k--; j--;
            }
        }
    }
    int max = count[0], index = 0;
    for (i = 1; i < k; i++) {
        if (count[i] > max) {
            max = count[i];
            index = i;
        }
    }

    printf("Most frequent word: '%s' (Count: %d)\n", words[index], max);
    return 0;
}
```

Input: the cat and the dog and the mouse
Output: Most frequent word: 'the' (Count: 3)

13.13 Lab Sheet 13: RE Acceptance

```
#include <stdio.h>
```

```
// Checks if string matches a*b (zero or more a's followed by b)
int match_a_star_b(char str[]) {
    int i = 0;
    while (str[i] == 'a') i++; // consume all a's
    return (str[i] == 'b' && str[i+1] == '\0');
}
int main() {
    char str[100];
    printf("Enter a string to check against RE: a*b\n");
    scanf("%s", str);

    if (match_a_star_b(str))
        printf("Accepted\n");
    else
        printf("Rejected\n");

    return 0;
}
```

Input: aaab

Output: Accepted

Input: b

Output: Accepted

Input: aac

Output: Rejected

All the Lab works are uploaded in Github-

Figure 13.1: Github-Compiler-design-lab-works

Appendix A

C Programming Essentials for Compiler Design

A.1 Strings, Arrays, Pointers Basics

A compiler often deals with textual data — source code written by a human programmer. That text is handled in a program using **strings**, which are usually implemented as arrays of characters in C.

1. Strings

In C, a string is an **array of characters terminated by a null character** `\0`.
Example:

```
char str[10] = "Hello";
```

Here:

- 'H' is stored in index 0
- 'e' in index 1
- 'l' in index 2
- 'l' in index 3
- 'o' in index 4
- '' in index 5

Memory representation of "Hello":

Index	0	1	2	3	4	5
Value	H	e	l	l	o	`\0`

Why important in compilers?

- The compiler reads source code as strings.
- Tokenization, parsing, and lexeme recognition rely heavily on string manipulation.

2. Arrays

An array is a collection of elements of the same type stored in contiguous memory.

Arrays can store:

- Characters (for strings)
- Integers (for symbol IDs, ASCII values, etc.)
- Structs (e.g., token tables)

Example:

```
int lineNumbers[100]; // stores up to 100 line numbers
```

In a compiler:

- Arrays are used to store symbol tables, intermediate code instructions, etc.

3. Pointers

A pointer is a variable that stores the memory address of another variable.
In compilers, pointers:

- Manage dynamic memory (symbol tables, syntax trees)
- Help in passing large data structures between functions
- Enable direct manipulation of source code strings

Example:

```
char *ptr;
char str[] = "Compiler";
ptr = str;
printf("%c", *ptr); // prints 'C'
```

4. Strings, Arrays, and Pointers Together

In C, strings and pointers are deeply related:

- `char str[] = "Hi";` stores the characters in an array.
- `char *p = str;` makes `p` point to the first character.
- `p + 1` points to the second character.

Example in Compiler Token Reading:

```
char source[] = "int a = 5;";
char *p = source;
while (*p != '\0') {
    printf("%c", *p);
    p++;
}
```

Practice Exercises

1. Store and print the characters of "Compiler Design" using an array and a pointer.
2. Implement a function to reverse a string manually.
3. Write a program to count vowels in a string without using library functions.

A.2 Common Input Techniques

Compilers must be able to read different kinds of input from a programmer's source file, such as:

- Single characters
- Strings
- Lines of text
- Entire files

A.2.1 1. Character Input

```
char ch;
ch = getchar(); // Reads one character
```

This technique is useful in lexical analysis, where the compiler reads the source code one character at a time.

A.2.2 2. String Input

```
char str[100];
scanf("%s", str); // Reads until whitespace
```

Limitation: This stops reading when a space is encountered, which is not ideal for reading full lines of source code.

A.2.3 3. Line Input

```
char line[100];
fgets(line, sizeof(line), stdin);
```

This method:

- Reads spaces
- Stops at a newline character or end-of-file

A.2.4 4. File Input

```
FILE *fp = fopen("source.c", "r");
char ch;
while ((ch = fgetc(fp)) != EOF) {
    putchar(ch);
}
fclose(fp);
```

File input is essential for compiling real program files. It allows the compiler to process an entire source code file directly from disk.

Practice Exercises

1. Read a program file and print all characters.
2. Read multiple lines until an empty line is entered.
3. Read a file and count how many times each keyword appears.

A.3 Manual vs Library Functions

When writing compiler-related programs, you often face a choice between using **library functions** and writing your own **manual implementations**.

A.3.1 Library Functions

Library functions are built-in, ready-to-use functions provided by the C standard library.

Examples: `strlen()`, `strcmp()`, `strcpy()`

Advantages:

- Saves development time.
- Usually optimized for performance.

Disadvantages:

- May hide the internal logic from beginners.
- Less control over specific implementation details.

A.3.2 Manual Implementation

Manual implementation means writing your own version of a function, handling all the logic yourself.

This approach:

- Forces a deeper understanding of how strings and memory work.
- Makes you aware of every operation performed.

Example: Implementing `strlen` manually

```
int myStrlen(char str[]) {
    int length = 0;
    while (str[length] != '\0') {
        length++;
    }
    return length;
}
```

A.3.3 In Compilers

For **educational projects**, manual methods are preferred to understand the underlying mechanisms.

For **real-world compilers**, optimized library functions are often used for speed and efficiency.

Appendix B

Reference Tables & Charts

B.1 ASCII Table

Char	Dec	Hex
A	65	41
a	97	61
0	48	30

B.2 Regex Symbol Table

Symbol	Meaning
*	Zero or more repetitions
+	One or more repetitions
?	Zero or one occurrence
‘ ‘	Escape sequence (special character)
()	Grouping

B.3 Grammar Notations

- Uppercase letters → Non-terminals
- Lowercase letters → Terminals
- → Production rule
- | Alternative

B.4 Token Classifications

- Keywords

- Identifiers
- Operators
- Literals
- Separators

B.5 Compiler Phases Recap

1. Lexical Analysis
2. Syntax Analysis
3. Semantic Analysis
4. Intermediate Code Generation
5. Optimization
6. Code Generation

Glossary

Lexeme: The smallest unit of meaning in the source code. Example: `if`, `x`, `42`.

Token: A category or classification of lexemes, such as `identifier`, `keyword`, or `literal`.

Grammar: A formal set of production rules that define the syntax of a language.

Bibliography

Bibliography

[1] A.V. Aho, M.S. Lam, R. Sethi, J.D. Ullman, *Compilers: Principles, Techniques, and Tools*, Pearson, 2006.

[2] C.N. Fischer, R.J. LeBlanc, *Crafting a Compiler*, Benjamin-Cummings, 1991.

About the Author

This handbook was written under the guidance of **Mr. Mushfiqur Rahman** Sir, during my Compiler Design coursework. The content reflects both theoretical and practical approaches, with emphasis on beginner-friendly explanations and detailed step-by-step programming examples. It is intended as a complete reference for students learning compiler construction from scratch.